

Arbitrary Precision Numbers

Petr Olšák

<ftp://math.feld.cvut.cz/olsak/makra/>

Table Of Contents

1	User's Documentation	2
1.1	Evaluation of Expressions	2
	<code>\evaldef</code> ... 2, <code>\apTOT</code> ... 3, <code>\apFRAC</code> ... 3, <code>\ABS</code> ... 3, <code>\SGN</code> ... 3, <code>\iDIV</code> ... 3,	
	<code>\iMOD</code> ... 3, <code>\iFLOOR</code> ... 3, <code>\iFRAC</code> ... 3, <code>\FAC</code> ... 3, <code>\BINOM</code> ... 3, <code>\SQRT</code> ... 3,	
	<code>\EXP</code> ... 3, <code>\LN</code> ... 3, <code>\SIN</code> ... 3, <code>\COS</code> ... 3, <code>\TAN</code> ... 3, <code>\ASIN</code> ... 3,	
	<code>\ACOS</code> ... 3, <code>\ATAN</code> ... 3, <code>\PI</code> ... 3, <code>\PIhalf</code> ... 3, <code>\OUT</code> ... 4, <code>\apSIGN</code> ... 4,	
	<code>\apE</code> ... 4	
1.2	Scientific Notation of Numbers	4
	<code>\apEadd</code> ... 4, <code>\apEnum</code> ... 4, <code>\apROLL</code> ... 5, <code>\apNORM</code> ... 5, <code>\apROUND</code> ... 5	
1.3	Notes for macro programmers	5
	<code>\apPLUS</code> ... 6, <code>\apMINUS</code> ... 6, <code>\apMUL</code> ... 6, <code>\apDIV</code> ... 6, <code>\apPOW</code> ... 6,	
	<code>\XOUT</code> ... 6	
1.4	Printing expressions	7
	<code>\eprint</code> ... 7, <code>\apMULop</code> ... 8, <code>\corrnum</code> ... 8	
1.5	Experiments	8
2	The Implementation	8
2.1	Name Convention, Version, Counters	8
	<code>\apVERSION</code> ... 9, <code>\apSIGN</code> ... 9, <code>\apE</code> ... 9, <code>\apTOT</code> ... 9, <code>\apFRAC</code> ... 9	
2.2	Evaluation of the Expression	9
	<code>\evaldef</code> ... 10, <code>\apEVALa</code> ... 10, <code>\OUT</code> ... 10, <code>\apEVALb</code> ... 10, <code>\apEVALc</code> ... 10,	
	<code>\apEVALd</code> ... 10, <code>\apEVALe</code> ... 10, <code>\apEVALf</code> ... 10, <code>\apEVALg</code> ... 10, <code>\apEVALh</code> ... 10,	
	<code>\apEVALk</code> ... 11, <code>\apEVALm</code> ... 11, <code>\apEVALn</code> ... 11, <code>\apEVALo</code> ... 11, <code>\apEVALp</code> ... 11,	
	<code>\apEVALstack</code> ... 11, <code>\apEVALpush</code> ... 12, <code>\apEVALdo</code> ... 12, <code>\apEVALerror</code> ... 12,	
	<code>\apTESTdigit</code> ... 12	
2.3	Preparation of the Parameter	12
	<code>\apPPa</code> ... 12, <code>\apPPb</code> ... 12, <code>\apPPc</code> ... 13, <code>\apPPd</code> ... 13, <code>\apPPe</code> ... 13, <code>\apPPf</code> ... 13,	
	<code>\apPPg</code> ... 13, <code>\apPPh</code> ... 13, <code>\apPPi</code> ... 13, <code>\apPPj</code> ... 13, <code>\apPPk</code> ... 13, <code>\apPPl</code> ... 13,	
	<code>\apPPm</code> ... 13, <code>\apPPn</code> ... 14, <code>\apPPab</code> ... 14, <code>\apPPs</code> ... 14, <code>\apPPt</code> ... 14, <code>\apPPu</code> ... 14	
2.4	Addition and Subtraction	14
	<code>\apPLUS</code> ... 15, <code>\apMINUS</code> ... 15, <code>\apPLUSa</code> ... 15, <code>\apPLUSxA</code> ... 15, <code>\apPLUSxB</code> ... 15,	
	<code>\apPLUSb</code> ... 16, <code>\apPLUSc</code> ... 16, <code>\apPLUSe</code> ... 16, <code>\apPLUSh</code> ... 17, <code>\apPLUSg</code> ... 17,	
	<code>\apPLUSd</code> ... 17, <code>\apPLUSf</code> ... 17, <code>\apPLUSm</code> ... 17, <code>\apPLUSp</code> ... 17, <code>\apPLUSw</code> ... 18,	
	<code>\apPLUSy</code> ... 18, <code>\apPLUSz</code> ... 18, <code>\apPLUSxE</code> ... 18	
2.5	Multiplication	18
	<code>\apMUL</code> ... 18, <code>\apMULa</code> ... 18, <code>\apMULb</code> ... 20, <code>\apMULc</code> ... 20, <code>\apMULd</code> ... 20,	
	<code>\apMULE</code> ... 20, <code>\apMULf</code> ... 20, <code>\apMULg</code> ... 21, <code>\apMULh</code> ... 21, <code>\apMULi</code> ... 21,	
	<code>\apMULj</code> ... 21, <code>\apMULo</code> ... 21, <code>\apMULt</code> ... 21	
2.6	Division	21
	<code>\apDIV</code> ... 23, <code>\apDIVa</code> ... 23, <code>\apDIVcomp</code> ... 24, <code>\apDIVcompA</code> ... 24, <code>\apDIVcompB</code> ... 24,	
	<code>\apDIVg</code> ... 25, <code>\apDIVh</code> ... 26, <code>\apDIVi</code> ... 26, <code>\apDIVj</code> ... 26, <code>\apDIVp</code> ... 26,	
	<code>\apDIVxA</code> ... 26, <code>\apDIVxB</code> ... 26, <code>\apDIVq</code> ... 26, <code>\apDIVr</code> ... 27, <code>\apDIVt</code> ... 27,	
	<code>\apDIVu</code> ... 27, <code>\XOUT</code> ... 28, <code>\apDIVv</code> ... 28, <code>\apDIVw</code> ... 28	
2.7	Power to the Integer	28
	<code>\apPOW</code> ... 28, <code>\apPOWx</code> ... 28, <code>\apPOWa</code> ... 28, <code>\apPOWb</code> ... 29, <code>\apPOWd</code> ... 29,	
	<code>\apPOWe</code> ... 30, <code>\apPOWg</code> ... 30, <code>\apPOWh</code> ... 30, <code>\apPOWn</code> ... 30, <code>\apPOWna</code> ... 30,	
	<code>\apPOWnn</code> ... 30, <code>\apPOWt</code> ... 30, <code>\apPOWu</code> ... 30, <code>\apPOWv</code> ... 30	

2.8	apROLL, apROUND and apNORM Macros	30
	<code>\apROLL... 30, \apROUND ...30, \apNORM ... 30, \apROLLa... 30, \apROLLc ...31,</code> <code>\apROLLd... 31, \apROLLe... 31, \apROLLf ...31, \apROLLg ...31, \apROLLh ... 31,</code> <code>\apROLLi... 31, \apROLLj... 32, \apROLLk ...32, \apROLLn ...32, \apROLLo ... 32,</code> <code>\apROUNDa ... 32, \apROUNDb ... 32, \apROUNDc ...32, \apROUNDd... 32, \apROUNDe ... 32,</code> <code>\apNORMa... 33, \apNORMb ...33, \apNORMc ... 33, \apNORMd ... 33, \apEadd... 33,</code> <code>\apEnum... 33</code>	
2.9	Miscellaneous Macros	33
	<code>\apEND ...33, \apDIG... 33, \apDIGa ... 33, \apDIGb ...33, \apDIGc ...34,</code> <code>\apDIGd... 34, \apDIGe ...34, \apDIGf ... 34, \apIVread ... 34, \apIVreadA ... 34,</code> <code>\apNL ...34, \apIVreadX... 34, \apIVwrite ... 34, \apIVtrans ...34, \apIVbase... 35,</code> <code>\apIVmod... 35, \apIVdot... 35, \apIVdotA ... 35, \apNUMdigits... 35,</code> <code>\apNUMdigitsA ...35, \apADDzeros... 35, \apREMzerosR ...35, \apREMzerosRa... 35,</code> <code>\apREMzerosRb ...35, \apREMDotR ...35, \apREMDotRa ...35, \apREMfirst ...35,</code> <code>\apOUTx... 36, \apOUTn ... 36, \apOUTl ...36, \apOUTs... 36, \apINIT ...36,</code> <code>\localcounts ...36, \apCOUNTS ...36, \do... 36, \apEVALxdo ... 36, \apRETURN ...36,</code> <code>\apERR ...36, \apNOPT ...37, \loop ...37</code>	
2.10	Function-like Macros	37
	<code>\ABS ...37, \SGN ...37, \iDIV... 37, \iMOD ...37, \iFLOOR ... 37, \iFRAC... 37,</code> <code>\FAC ...37, \BINOM ...38, \SQRT ...38, \apSQRTxo... 39, \apSQRTTr ...39,</code> <code>\apSQRTRa ... 39, \apSQRTTrb... 39, \EXP ...40, \apTAYLOR ...40, \LN ... 41,</code> <code>\apLNtaylor ...42, \apLNr ... 42, \apLNra ... 42, \apLNrten ...42, \apLNtenexec ... 42,</code> <code>\apLNten... 42, \apPiValue ... 43, \apPidigits ...43, \apPiexec... 43, \apPI ...43,</code> <code>\apPiHalf ... 43, \apPiexecA ... 43, \apPiexecB ... 43, \PI ... 44, \PiHalf... 44,</code> <code>\SIN ...44, \COS ... 44, \apSINCOSa... 44, \apSINx... 46, \apCOSx ...46,</code> <code>\apSINCOSo ... 46, \TAN... 46, \ATAN ...46, \apATANox... 47, \ASIN... 47, \ACOS ...47</code>	
2.11	Printing expressions	47
	<code>\eprint... 47, \apEpe... 47, \apEpi ... 47, \apEPj ...47, \apEPplus ... 48,</code> <code>\apEPminus ... 48, \apEPmul ... 48, \apEPdiv ... 48, \apEppow... 48, \apEPy ... 48,</code> <code>\apEppowa ... 49, \apEPf ... 49, \apEPb ... 49, \apEP ...49, \apEPa ...49,</code> <code>\apMULop... 49, \apEPabs... 49, \apEPfac ... 49, \apEPbinom ...49, \apEPsqrt ...49,</code> <code>\apEPexp... 50, \apEPsgn... 50, \apEPdivmod ...50, \apEPidiv... 50, \apEPimod... 50,</code> <code>\apEPifloor ...50, \apEPifrac... 50, \corrnum ... 50, \apEPc ... 50</code>	
2.12	Conclusion	50
3	Index	50

1 User's Documentation

This macro file `apnum.tex` implements addition, subtraction, multiplication, division, power to an integer and other calculation (\sqrt{x} , e^x , $\ln x$, $\sin x$, $\arctan x$, ...) with “large numbers” with arbitrary number of decimal digits. The numbers are in the form:

`<sign><digits>.<digits>`

where optional $\langle sign \rangle$ is the sequence of + and/or -. The nonzero number is treated as negative if and only if there is odd number of - signs. The first part or second part of decimal $\langle digits \rangle$ (but no both) can be empty. The decimal point is optional if second part of $\langle digits \rangle$ is empty.

There can be unlimited number of digits in the operands. Only TeX main memory or your patience during calculation with very large numbers are your limits. Note, that the `apnum.tex` implementation includes a lot of optimization and it is above 100 times faster (on large numbers) than the implementation of the similar task in the package `fltpoint.sty`. And the `fp.sty` doesn't implements arbitrary number of digits. The extensive technical documentation can serve as an inspiration how to do TeX macro programming.

1.1 Evaluation of Expressions

After `\input apnum` in your document you can use the macro `\evaldef<sequence>\{<expression>\}`. It gives the possibility for comfortable calculation. The $\langle expression \rangle$ can include numbers (in the form

described above) combined by +, -, *, / and ^ operators and by possible brackets () in an usual way. The result is stored to the $\langle sequence \rangle$ as a “literal macro”. Examples:

```
\evaldef\A {2+4*(3+7)}
% ... the macro \A includes 42
\evaldef\B {\the\pageno * \A}
% ... the macro \B includes 84
\evaldef\C {123456789000123456789 * -123456789123456789123456789}
% ... \C includes -15241578765447341344197531849955953099750190521
\evaldef\D {1.23456789 + 12345678.9 - \A}
% ... the macro \D includes 12345596.13456789
\evaldef\X {1/3}
% ... the macro \X includes .333333333333333333
```

The limit of the number of digits of the division result can be set by `\apTOT` and `\apFRAC` registers. First one declares maximum calculated digits in total and second one declares maximum of digits after decimal point. The result is limited by both those registers. If the `\apTOT` is negative, then its absolute value is treated as a “soft limit”: all digits before decimal point are calculated even if this limit is exceeded. The digits after decimal point are not calculated when this limit is reached. The special value `\apTOT=0` means that the calculation is limited only by `\apFRAC`. Default values are `\apTOT=0` and `\apFRAC=20`.

The operator ^ means the powering, i. e. 2^8 is 256. The exponent have to be an integer (no decimal point is allowed) and a relatively small integer is assumed.

The scanner of the `\evaldef` macro reads (roughly speaking) the $\langle expression \rangle$ in the form “operand binary-operator operand binary-operator etc.” without expansion. The spaces are not significant in the $\langle expression \rangle$. The operands are:

- numbers (in the format $\langle sign \rangle \langle digits \rangle . \langle digits \rangle$) or
- numbers in scientific notation (see the section 1.2) or
- sequences $\langle sign \rangle \backslash the \langle token \rangle$ or $\langle sign \rangle \backslash number \langle token \rangle$ or
- any other single $\langle token \rangle$ optionally preceded by $\langle sign \rangle$ and optionally followed by a sequence of parameters enclosed in braces, for example `\A` or `\B{ $\langle text \rangle$ }` or `-\C{{ $\langle textA \rangle$ }}{{ $\langle textB \rangle$ }}`. This case has two meanings:
 - numeric constant defined in a “literal macro” (`\def\A{42}`, `\evaldef\A{13/15}`) or
 - “function-like” macro which returns a value after processing.

The `apnum.tex` macro file provides the following “function-like” macros allowed to use them as an operand in the $\langle expression \rangle$:

- `\ABS { $\langle value \rangle$ }` for absolute value,
- `\SGN { $\langle value \rangle$ }` returns sign of the $\langle value \rangle$,
- `\iDIV { $\langle dividend \rangle$ }{ $\langle divisor \rangle$ }` for integer division,
- `\iMOD { $\langle dividend \rangle$ }{ $\langle divisor \rangle$ }` for integer remainder,
- `\iFLOOR { $\langle value \rangle$ }` for rounding the number to the integer,
- `\iFRAC { $\langle value \rangle$ }` for fraction part of the `\iFLOOR`,
- `\FAC { $\langle integer value \rangle$ }` for factorial,
- `\BINOM { $\langle integer above \rangle$ }{ $\langle integer below \rangle$ }` for binomial coefficient,
- `\SQRT { $\langle value \rangle$ }` for square root of the $\langle value \rangle$,
- `\EXP { $\langle value \rangle$ }` applies exponential function to $\langle value \rangle$,
- `\LN { $\langle value \rangle$ }` for natural logarithm of the $\langle value \rangle$,
- `\SIN { $\langle value \rangle$ }`, `\COS { $\langle value \rangle$ }`, `\TAN { $\langle value \rangle$ }` for $\sin x$, $\cos x$ and $\tan x$ functions,
- `\ASIN { $\langle value \rangle$ }`, `\ACOS { $\langle value \rangle$ }`, `\ATAN { $\langle value \rangle$ }` for $\arcsin x$, $\arccos x$ and $\arctan x$ functions,
- `\PI`, `\PIhalf` for constants π and $\pi/2$.

The arguments of all these functions can be a nested $\langle expressions \rangle$ with the syntax like in the `\evaldef` macro. Example:

```
\def\A{20}
\evaldef\B{ 30*\SQRT{ 100 + 1.12*\the\widowpenalty } / (4-\A) }
```

Note that the arguments of the “function-like” macros are enclosed by normal TeX braces {} but the round brackets () are used for re-arranging of the common priority of the +, -, *, / and ^ operators. The macros `\SQRT`, `\EXP`, `\LN`, `\SIN`, `\COS`, `\TAN`, `\ASIN`, `\ACOS`, `\ATAN` use `\apTOT` and `\apFRAC` registers similar like during division.

The `\PI` and `\PIhalf` are “function-like” macros without parameters. They returns the constant with `\apFRAC` digits after decimal point.

Users can define their own “function-like” macros, see the section 1.3.

The output of `\evaldef\foo{<expression>}` processing is stored, of course, to the “literal macro” `\foo`. But there are another outputs like side effect of the processing:

- The `\OUT` macro includes exactly the same result as `\foo`.
- The `\apSIGN` register includes the value 1 or 0 or -1 respectively dependent on the fact that the output is positive, zero or negative.
- The `\apE` register is equal to the decimal exponent when scientific number format is used (see the next section 1.2).

For example, you can compare long numbers using `\apSIGN` register (where direct usage of `\ifnum` primitive may cause arithmetic overflow):

```
\TEST {123456789123456789} > {123456789123456788} \iftrue OK \else KO \fi
```

The `\TEST` macro is defined like:

```
\def\TEST#1#2#3#4{\evaldef\tmp{#1-(#3)}\ifnum\apSIGN #2 0 }
```

The `apnum.tex` macros do not provide the evaluation of the `<expression>` at the expansion level only. There are two reasons. First, the macros can be used in classical TeX only with Knuth's plain TeX macro. No eTeX is needed. And the expansion-only evaluation of any expression isn't possible in classical TeX. Second reason is the speed optimization (see the section 1.5). Anyway, users needn't expansion-only evaluation. They can write `\evaldef\aa{<expression>} \edef\foo{... \aa...}` instead of `\edef\foo{...<expression>...}`. There is only one case when this “pre-processing” trick cannot be used: while expansion of the parameters of asynchronous `\write` commands. But you can save the `<expression>` unexpanded into the file and you can read the file again in the second step and do `\evaldef` during reading the file.

1.2 Scientific Notation of Numbers

The macro `\evaldef` is able to operate with the numbers written in the notation:

```
<sign><digits>.<digits>E<sign><digits>
```

For example `1.234E9` means $1.234 \cdot 10^9$, i. e. 1234000000 or the text `1.234E-3` means .001234. The decimal exponent (after the E letter) have to be in the range ± 2147483647 because we store this value in normal TeX register.

The `\evaldef<sequence>{<expression>}` operates by “normal way” if there are no operands with E syntax in the `<expression>`. But if an operand is expressed in scientific form then `\evaldef` provide the calculation with the mantissa and the exponent separately. Only the mantissa of the result is found in the `<sequence>` and `\OUT` macros. The exponent of the result is stored in the `\apE` register. You can define the macro which shows the complete result of the calculation, for example:

```
\def\showE#1{\message{#1\ifnum\apE=0 \else*10^{\the\apE}\fi}}
```

Suppose `\evaldef\foo{<expression>}` is processed and the complete result is $R = \text{\foo} * 10^{\text{\apE}}$. There are two possibilities how to save such complete result R to the `\foo` macro: use `\apEadd\foo` or `\apEnum\foo`. Both macros do nothing if `\apE=0`. Else the `\apEadd<sequence>` macro adds $E<exponent>$ to the `<sequence>` macro and `\apEnum<sequence>` moves the decimal point to the new right position in the `<sequence>` macro or appends zeros. The `\apE` register is set to zero after the macro `\apEadd` or `\apEnum` is finished. Example:

```

\evaldef\foo{ 3 * 4E9 }      % \foo is 12, \apE=9
\apEadd\foo                  % \foo is 12E+9
\evaldef\foo{ 7E9 + 5E9 }    % \foo is 12, \apE=9
\apEnum\foo                  % \foo is 12000000000

```

There are another usable macros for operations with scientific numbers.

- **\apROLL** $\langle sequence \rangle \{ \langle shift \rangle \}$... the $\langle sequence \rangle$ is assumed to be a macro with the number. The decimal point of this number is shifted right by $\langle shift \rangle$ parameter, i. e. the result is multiplied by $10^{\langle shift \rangle}$. The $\langle sequence \rangle$ is redefined by this result. For example the **\apEnum**\A does **\apROLL**\A{\apE}.
- **\apNORM** $\langle sequence \rangle \{ \langle num \rangle \}$... the $\langle sequence \rangle$ is supposed to be a macro with $\langle mantissa \rangle$ and it will be redefined. The number $\langle mantissa \rangle * 10^{\langle apE \rangle}$ (with current value of the **\apE** register) is assumed. The new mantissa saved in the $\langle sequence \rangle$ is the “normalized mantissa” of the same number. The **\apE** register is corrected so the “normalized mantissa” * $10^{\langle apE \rangle}$ gives the same number. The $\langle num \rangle$ parameter is the number of non-zero digits before the decimal point in the outputted mantissa. If the parameter $\langle num \rangle$ starts by dot following by integer (for example {.2}), then the outputted mantissa has $\langle num \rangle$ digits after decimal point. For example **\def**\A{1.234}\apE=0 **\apNORM**\A{.0} defines \A as 1234 and **\apE**=-3.
- The **\apROUND** $\langle sequence \rangle \{ \langle num \rangle \}$ rounds the number, which is included in the macro $\langle sequence \rangle$ and redefines $\langle sequence \rangle$ as rounded number. The digits after decimal point at the position greater than $\langle num \rangle$ are ignored in the rounded number. The decimal point is removed, if it is the right most character in the **\OUT**. The ignored part is saved to the **\XOUT** macro without trailing right zeros.

Examples of **\apROUND** usage:

```

\def\A{12.3456}\apROUND\A{1} % \A is "12.3", \XOUT is "456"
\def\A{12.3456}\apROUND\A{9} % \A is "12.3456", \XOUT is empty
\def\A{12.3456}\apROUND\A{0} % \A is "12", \XOUT is "3456"
\def\A{12.0000}\apROUND\A{0} % \A is "12", \XOUT is empty
\def\A{12.0001}\apROUND\A{2} % \A is "12", \XOUT is "01"
\def\A{.000010}\apROUND\A{2} % \A is "0", \XOUT is "001"
\def\A{-12.3456}\apROUND\A{2} % \A is "-12.34", \XOUT is "56"
\def\A{12.3456}\apROUND\A{-1} % \A is "10", \XOUT is "23456"
\def\A{12.3456}\apROUND\A{-4} % \A is "0", \XOUT is "00123456"

```

The following example saves the result of the **\evaldef** in scientific notation with the mantissa with maximal three digits after decimal point and one digit before.

```

\evaldef\X{...}\apNORM\X{1}\apROUND\X{3}\apEadd\X

```

The macros **\apEadd**, **\apEnum**, **\apROLL**, **\apNORM** and **\apROUND** redefine the macro $\langle sequence \rangle$ given as their first argument. They are not “function-like” macros and they cannot be used in an $\langle expression \rangle$. The macro $\langle sequence \rangle$ must be the number in the format $\langle simple\ sign \rangle \langle digits \rangle . \langle digits \rangle$ where $\langle simple\ sign \rangle$ is one minus or none and the rest of number has the format described in the first paragraph of this documentation. The scientific notation isn’t allowed here. This format of numbers is in accordance with the output of the **\evaldef** macro.

1.3 Notes for macro programmers

If you plan to create a “function-like” macro which can be used as an operand in the $\langle expression \rangle$ then observe that first token in the macro body must be **\relax**. This tells to the $\langle expression \rangle$ scanner that the calculation follows. The result of this calculation must be saved into the **\OUT** macro and into the **\apSIGN** and **\apE** registers.

Example. The **\ABS** macro for the absolute value is defined by:

```

704: \def\ABS#1{\relax          % mandatory \relax for "function-like" macros
705:   \evaldef\OUT{#1}%        % evaluation of the input parameter
706:   \ifnum\apSIGN<0          % if (input < 0)
707:     \apSIGN=1               % sign = 1

```

apnum.tex

```

708:      \apREMfirst\OUT      %   remove first "minus" from OUT
709:      \fi                  % fi

```

```
Usage: \evaldef\A{ 2 - \ABS{3-10} }% \A includes -5.
```

Note, that `\apSIGN` register is corrected by final routine of `\evaldef` according the `\OUT` value. But setting `\apSIGN` in your macro is recommended because user can use your macro directly outside of `\evaldef`.

The `\evaldef\foo{<expression>}` is processed in two steps. The `<expression>` scanner converts the input to the macro call of the `\apPLUS`, `\apMINUS`, `\apMUL`, `\apDIV` or `\apPOW` macros with two parameters. They do addition, subtraction, multiplication, division and power to the integer. These macros are processed in the second step. For example:

```

\evaldef\A{ 2 - 3*8 }      converts the input to:
\apMINUS{2}{\apMUL{3}{8}} and this is processed in the second step.

```

The macros `\apPLUS`, `\apMINUS`, `\apMUL`, `\apDIV` and `\apPOW` behave like normal “function-like” macros with one important exception: they don’t accept general `<expression>` in their parameters, only single operand (see section 1.1) is allowed.

If your calculation is processed in the loop very intensively than it is better to save time of such calculation and to avoid the `<expression>` scanner processing (first step of the `\evaldef`). So, it is recommended to use directly the Polish notation of the expression as shown in the second line in the example above. See section 2.10 for more inspirations.

The output of the `\apPLUS`, `\apMINUS`, `\apMUL`, `\apDIV` and `\apPOW` macros is stored in `\OUT` macro and the registers `\apSIGN` and `\apE` are set accordingly.

The number of digits calculated by `\apDIV` macro is limited by the `\apTOT` and `\apFRAC` registers as described in the section 1.1. There is another result of `\apDIV` calculation stored in the `\XOUT` macro. It is the remainder of the division. Example:

```
\apTOT=0 \apFRAC=0 \apDIV{1234567892345}{2}\ifnum\XOUT=0 even \else odd\fi
```

You cannot apply `\ifodd` primitive on “large numbers” directly because the numbers may be too big.

If you set something locally inside your “function-like” macro, then such data are accessible only when your macro is called outside of `\evaldef`. Each parameter and the whole `\evaldef` is processed inside a \TeX group, so your locally set data are inaccessible when your macro is used inside another “function-like” parameter or inside `\evaldef`. The `\XOUT` output is set locally by `\apDIV` macro, so it serves as a good example of this feature:

```

{\apDIV{1}{3}          ... \XOUT is .00000000000000000001 }
                        ... \XOUT is undefined
\evaldef{1/3}          ... \XOUT is undefined
\apPLUS{1}{\apDIV{1}{3}} ... \XOUT is undefined

```

The macro `\apPOW{<base>}{<exponent>}` calculates the power to the integer exponent. A slight optimization is implemented here so the usage of `\apPOW` is faster than repeated multiplication. The decimal non-integer exponents are not allowed. Use `\EXP` and `\LN` macros if you need to calculate non-integer exponent:

```
\def\POWER#1#2{\relax \EXP{(#2)*\LN{#1}}}
```

Note that both parameters are excepted as an `<expression>`. Thus the `#2` is surrounded in the rounded brackets.

Examples of another common “function-like” macros:

```

\evaldef\degcoef{PI/180}
\def\SINdeg#1{\relax \SIN{\degcoef*(#1)}}
\def\COSdeg#1{\relax \COS{\degcoef*(#1)}}
\def\SINH#1{\relax \evaldef\myE{\EXP{#1}}\evaldef\OUT{(\myE-1/\myE)/2}}
\def\ASINH#1{\relax \LN{#1+\SQRT{(#1)^2+1}}}
\def\LOG#1{\relax \apLNtenexec \apDIV{\LN{#1}}{\apLNten}}

```


In another example, we implement the field $\backslash F\{\langle index \rangle\}$ as an “function-like” macro. User can set values by $\backslash set \backslash F\{\langle index \rangle\} = \{\langle value \rangle\}$ and then these values can be used in an $\langle expression \rangle$.

```
\def\set#1#2#3#4{\evaldef\index{#2}\evaldef\value{#4}%
  \expandafter\edef\csname \string#1[\index]\endcsname{\value}}
\def\F#1{\relax % function-like macro
  \evaldef\index{#1}%
  \expandafter\ifx\csname \string\F[\index]\endcsname\relax
    \def\OUT{0}% undefined value
  \else
    \edef\OUT{\csname \string\F[\index]\endcsname}%
  \fi
}
\set \F{12/2} = {28+13}
\set \F{2*4} = {144^2}
\evaldef\test { 1 + \F{6} } \message{result=\test}
```

As an exercise, you can implement linear interpolation of known values.

The final example shows, how to implement the macro $\backslash usedimen\{\langle dimen \rangle\}\{\langle unit \rangle\}$. It is “function-like” macro, it can be used in the $\langle expression \rangle$ and it returns the $\langle decimal number \rangle$ with the property $\langle dimen \rangle = \langle decimal number \rangle \langle unit \rangle$.

```
\def\usedimen #1#2{\relax % function-like macro
  \def\OUT{0}% % default value, if the unit isn't known
  \csname dimenX#2\endcsname{#1}}
\def\dimenXpt #1{\apDIV{\number#1}{65536}}
\def\dimenXcm #1{\apDIV{\number#1}{1864682.7}}
\def\dimenXmm #1{\apDIV{\number#1}{186468.27}}
%... etc.
\evaldef\aa{\usedimen{\hsize}{cm}} % \aa includes 15.91997501773358008845
```

Note that user cannot write $\backslash usedimen\hsize\{cm\}$ without braces because this isn't the syntactically correct operand (see section 1.1) and the $\langle expression \rangle$ scanner is unable to read it.

1.4 Printing expressions

TeX was designed for printing. The `apnum.tex` provides common syntax of $\langle expressions \rangle$ (given in section 1.1) which can be used for both: for *evaluating* or for *printing*. Printing can be done using $\backslash eprint\{\langle expression \rangle\}\{\langle declaration \rangle\}$ macro. The $\langle declaration \rangle$ part declares locally what to do with “variables” or with your “function-like” macros. You can insert your local $\backslash def$'s or $\backslash let$'s here because the $\langle declaration \rangle$ is executed in the group before the $\langle expression \rangle$ is printed. The $\backslash eprint$ macro must be used in math mode only. Example:

```
\def\printresult#1{$$\displaylines{
  \eprint{#1}\vars = \cr = \eprint{#1}\nums = \cr
  = \apFRAC=8 \evaldef\OUT{#1}\OUT, \cr
  \nums x = \X, \quad y = \Y.
}$$}

\def\X{-.25} \def\Y{18.11}
\def\vars{\def\X{x}\def\Y{y}\let\apMULop=\relax}
\def\nums{\corrnum\X \corrnum\Y}
\printresult
{-(\X-\SQRT{\Y^2+1}) + -((\X*\Y+1)/2) + \SIN{\X+\PIhalf} + 2*\COS{\Y}}
```

generates the result:

$$\begin{aligned}
 & -\left(x - \sqrt{y^2 + 1}\right) + -\frac{xy + 1}{2} + \sin\left(x + \frac{\pi}{2}\right) + 2 \cos y = \\
 & = -\left(-0.25 - \sqrt{18.11^2 + 1}\right) + -\frac{-0.25 \cdot 18.11 + 1}{2} + \sin\left(-0.25 + \frac{\pi}{2}\right) + 2 \cdot \cos 18.11 = \\
 & \quad = 22.5977863, \\
 & \quad x = -0.25, \quad y = 18.11
 \end{aligned}$$

This example prints the given $\langle expression \rangle$ in two forms: with “variables as variables” first and with “variables as constants” second. The $\langle declaration \rangle$ is prepared in the `\vars` macro for the first form and in the `\nums` macro for the second.

Note that `\eprint` macro re-calculates the occurrences of round brackets but keeps the meaning of the $\langle expression \rangle$. For example $(\A+\B)/\C$ is printed as `{a+b\over c}` (without brackets) and $6*-(\A+\B)$ is printed as `6\cdot(-(a+b))` (new brackets pair is added). Or `\SIN{\X}` is printed as `\sin x` (without brackets) but `\SIN{\X+1}` is printed as `\sin(x+1)` (with brackets). And `\SIN{\X}^2` is printed as `\sin^2 x`.

You can do `\let\apMULop=`, or `\let\apMULop=\relax` in the $\langle declaration \rangle$ part if you need not to print any operator for multiplying. The default setting is `\let\apMULop=\cdot`. Another possibility is to set `\let\apMULop=\times`.

The macro `\corrnum<token>` corrects the number saved in the $\langle token \rangle$ macro if it is in the form $[\langle minus \rangle] . \langle digits \rangle$ (i. e. without digits before decimal point). Then zero is added before decimal point. Else nothing is changed.

Warning. The first parameter of `\eprint` (i. e. the $\langle expression \rangle$), must be directly expression without any expansion steps. For example, you cannot define `\def\foo{\langle expression \rangle}` and do `\eprint{\foo}` but you can do `\expandafter\eprint\expandafter{\foo}`.

The macro `\eprint` has its own intelligence about putting brackets. If you need to put or remove brackets somewhere where the intelligence of `\eprint` is different from your opinion, you can create your function-like macros `\BK{\langle expression \rangle}` and `\noBK{\langle expression \rangle}`. They evaluate the $\langle expression \rangle$ when using `\evaldef`. The `\BK` prints the $\langle expression \rangle$ with brackets and `\noBK` prints it without brackets when using `\eprint`.

```

\def\BK#1{\relax \evaldef\OUT{#1}}
\let\noBK=\BK
\def\BKprint#1{\left(\eprint{#1}\right)}
\def\noBKprint#1{\eprint{#1}}
\def\setBK{\let\BK=\BKprint \let\noBK=\noBKprint}
Now $\eprint{3+\BK{\SIN{1}}^2}\setBK$ prints $3+(\sin 1)^2$.

```

1.5 Experiments

The following table shows the time needed for calculation of randomly selected examples. The comparison with the package `fltpoint.sty` is shown. The symbol ∞ means that it is out of my patience.

input	# of digits in the result	time spent by <code>apnum.tex</code>	time spent by <code>fltpoint.sty</code>
200!	375	0.33 sec	173 sec
1000!	2568	29 sec	∞
5^{17^2}	203	0.1 sec	81 sec
5^{17^3}	3435	2.1 sec	∞
1/17	1000	0.13 sec	113 sec
1/17	100000	142 sec	∞

2 The Implementation

2.1 Name Convention, Version, Counters

The internal control sequence names typically used in `apnum.tex` have the form `\apNAMEsuffix`, but there are exceptions. The control sequences mentioned in the section 1.1 (user’s documentation)

have typically more natural names. And the internal counter registers have names `\apnumA`, `\apnumB`, `\apnumC` etc.

The code starts by the greeting. The `\apVERSION` includes the version of this software.

```
7: \def\apVERSION{1.4a <Dec 2015>}
8: \message{The Arbitrary Precision Numbers, \apVERSION}
```

apnum.tex

We declare auxiliary counters and one Boolean variable.

```
12: \newcount\apnumA \newcount\apnumB \newcount\apnumC \newcount\apnumD
13: \newcount\apnumE \newcount\apnumF \newcount\apnumG \newcount\apnumH
14: \newcount\apnumO \newcount\apnumP \newcount\apnumL
15: \newcount\apnumX \newcount\apnumY \newcount\apnumZ
16: \newcount\apSIGNa \newcount\apSIGNb \newcount\apEa \newcount\apEb
17: \newif\ifapX
```

apnum.tex

The counters `\apSIGN`, `\apE`, `\apTOT` and `\apFRAC` are declared here:

```
19: \newcount\apSIGN
20: \newcount\apE
21: \newcount\apTOT \apTOT=0
22: \newcount\apFRAC \apFRAC=20
```

apnum.tex

Somebody sometimes sets the @ character to the special catcode. But we need to be sure that there is normal catcode of the @ character.

```
24: \apnumZ=\catcode'\@ \catcode'\@=12
```

apnum.tex

2.2 Evaluation of the Expression

Suppose the following expression $A+B*(C+D)+E$ as an example.

The main task of the `\evaldef` $x\{A+B*(C+D)+E\}$ is to prepare the macro `\tmpb` with the content (in this example) `\apPLUS{\apPLUS{\A}{\apMUL{\B}{\apPLUS{\C}{\D}}}}{\E}` and to execute the `\tmpb` macro.

The expression scanner adds the `\limits` at the end of the expression and reads from left to right the couples “operand, operator”. For our example: `\A+`, `\B*`, `\C+`, `\D+` and `\E\limits`. The `\limits` operator has the priority 0, plus, minus have priority 1, `*` and `/` have priority 2 and `^` has priority 3. The brackets are ignored, but each occurrence of the opening bracket `(` increases priority by 4 and each occurrence of closing bracket `)` decreases priority by 4. The scanner puts each couple including its current priority to the stack and does a test at the top of the stack. The top of the stack is executed if the priority of the top operator is less or equal the previous operator priority. For our example the stack is only pushed without execution until `\D+` occurs. Our example in the stack looks like:

<code>\D + 1</code>	<code>1<=5 exec:</code>	
<code>\C + 5</code>	<code>{\C+\D} + 1</code>	<code>1<=2 exec:</code>
<code>\B * 2</code>	<code>\B * 2</code>	<code>{\B*{\C+\D}} + 1</code>
<code>\A + 1</code>	<code>\A + 1</code>	<code>\A + 1</code>
<code>bottom 0</code>	<code>bottom 0</code>	<code>bottom 0</code>

Now, the priority on the top is greater, then scanner pushes next couple and does the test on the top of the stack again.

<code>\E \limits 0</code>	<code>0<=1 exec:</code>	
<code>{\A+{\B*{\C+\D}}} + 1</code>	<code>{{\A+{\B*{\C+\D}}}+\E} \end 0</code>	<code>0<=0 exec:</code>
<code>bottom 0</code>	<code>bottom 0</code>	RESULT

Let p_t, p_p are the priority on the top and the previous priority in the stack. Let v_t, v_p are operands on the top and in the previous line in the stack, and the same notation is used for operators o_t and o_p . If $p_t \leq p_p$ then: pop the stack twice, create composed operand $v_n = v_p o_p v_t$ and push v_n, o_t, p_t . Else push new couple “operand, operator” from the expression scanner. In both cases try to execute the top of the stack again. If the bottom of the stack is reached then the last operand is the result.

`\apVERSION`: 9 `\apSIGN`: 4, 5–6, 9–10, 12–16, 18–19, 23–24, 29, 33, 37–42, 45–47, 50
`\apE`: 4, 5–6, 9–14, 16, 18–19, 23–24, 28–29, 33, 37, 39, 41 `\apTOT`: 3, 4, 6, 9, 24, 37, 43–46
`\apFRAC`: 3, 4, 6, 9, 24, 37, 39–47

The `\evaldef` macro is protected by `\relax`. It means that it can be used inside an $\langle expression \rangle$ as a “function-like” macro, but I don’t imagine any usual application of this. The `\apEVALa` is executed. apnum.tex

```
28: \def\evaldef\relax \apEVALa}
```

The macro `\apEVALa` $\langle sequence \rangle \{ \langle expression \rangle \}$ runs the evaluation of the expression in the group. The base priority is initialized by `\apnumA=0`, then `\apEVALb` $\langle expression \rangle \backslash limits$ scans the expression and saves the result in the form `\apPLUS{\A}{\apMUL{\B}{\C}}` (etc.) into the `\tmpb` macro. This macro is executed. The group is finished by `\apEND` macro, which keeps the `\OUT`, `\apSIGN` and `\apE` values unchanged. Finally the defined $\langle sequence \rangle$ is set equivalent to the `\OUT` macro.

```
29: \def\apEVALa#1#2{\begingroup \apnumA=0 \apnumE=1 \apEVALb#2\limits \tmpb \apEND \let#1=\OUT}
```

The scanner is in one of the two states: reading operand or reading operator. The first state is initialized by `\apEVALb` which follows to the `\apEVALc`. The `\apEVALc` reads one token and switches by its value. If the value is a + or - sign, it is assumed to be the part of the operand prefix. Plus sign is ignored (and `\apEVALc` is run again), minus signs are accumulated into `\tmpa`.

The auxiliary macro `\apEVALd` runs the following tokens to the `\fi`, but first it closes the conditional and skips the rest of the macro `\apEVALc`.

```
30: \def\apEVALb{\def\tmpa{}\apEVALc}
31: \def\apEVALc#1{%
32:   \ifx+#1\apEVALd \apEVALc \fi
33:   \ifx-#1\edef\tmpa{\tmpa-}\apEVALd\apEVALc \fi
34:   \ifx(#1\apEVALd \apEVALe \fi
35:   \ifx\the#1\apEVALd \apEVALf\the\fi
36:   \ifx\number#1\apEVALd \apEVALf\number\fi
37:   \apTESTdigit#1\iftrue
38:     \ifx E#1\let\tmpb=\tmpa \expandafter\apEVALd\expandafter\apEVALk
39:     \else \edef\tmpb{\tmpa#1}\expandafter\apEVALd\expandafter\apEVALn\fi\fi
40:   \edef\tmpb{\tmpa\noexpand#1}\expandafter
41:   \futurelet\expandafter\apNext\expandafter\apEVALg\romannumeral-'\.%
42: }
43: \def\apEVALd#1\fi#2-'\.{\fi#1}
```

If the next token is opening bracket, then the global priority is increased by 4 using the macro `\apEVALe`. Moreover, if the sign before bracket generates the negative result, then the new multiplication (by -1) is added using `\apEVALp` to the operand stack.

```
44: \def\apEVALe{%
45:   \ifx\tmpa\empty \else \ifnum\tmpa1<0 \def\tmpb{-1}\apEVALp \apMUL 4\fi\fi
46:   \advance\apnumA by4
47:   \apEVALb
48: }
```

If the next token is `\the` or `\number` primitives (see lines 35 and 36), then one following token is assumed as TeX register and these two tokens are interpreted as an operand. This is done by `\apEVALf`. The operand is packed to the `\tmpb` macro.

```
49: \def\apEVALf#1#2{\expandafter\def\expandafter\tmpb\expandafter{\tmpa#1#2}\apEVALo}
```

If the next token is not a number (the `\apTESTdigit#1\iftrue` results like `\iffalse` at line 37) then we save the sign plus this token to the `\tmpb` at line 41 and we do check of the following token by `\futurelet`. The `\apEVALg` is processed after that. The test is performed here if the following token is open brace (a macro with parameter). If this is true then this parameter is appended to `\tmpb` by `\apEVALh` and the test about the existence of second parameter in braces is repeated by next `\futurelet`. The result of this loop is stored into `\tmpb` macro which includes $\langle sign \rangle$ followed by $\langle token \rangle$ followed by all parameters in braces. This is considered as an operand.

```
50: \def\apEVALg{\ifx\apNext \bgroup \expandafter\apEVALh \else \expandafter\apEVALo \fi}
51: \def\apEVALh#1{\expandafter\def\expandafter\tmpb\expandafter{\tmpb#1}}\expandafter
```

`\evaldef`: 2, 3–6, 8–10, 12, 28, 36–41, 45–47 `\apEVALa`: 10, 12 `\OUT`: 4, 5–6, 10, 12–14, 16–21,
23–24, 26–30, 33, 36–47 `\apEVALb`: 10–11, 47 `\apEVALc`: 10–11 `\apEVALd`: 10 `\apEVALe`: 10
`\apEVALf`: 10 `\apEVALg`: 10 `\apEVALh`: 10

If the next token after the sign is a digit or a dot (tested in `\apEVALc` by `\apTESTdigit` at line 37), then there are two cases. The number includes the E symbol as a first symbol (this is allowed in scientific notation, mantissa is assumed to equal to one). The `\apEVALk` is executed in such case. Else the `\apEVALn` starts the reading the number.

The first case with E letter in the number is solved by macros `\apEVALk` and `\apEVALm`. The number after E is read by `\apE=` and this number is appended to the `\tmpb` and the expression scanner skips to `\apEVALo`.

apnum.tex

```
53: \def\apEVALk{\afterassignment\apEVALm\apE=}
54: \def\apEVALm{\edef\tmpb{\tmpb E\the\apE}\apEVALo}
```

The second case (there is normal number) is processed by the macro `\apEVALn`. This macro reads digits (token per token) and saves them to the `\tmpb`. If the next token isn't digit nor dot then the second state of the scanner (reading an operator) is invoked by running `\apEVALo`. If the E is found then the exponent is read to `\apE` and it is processed by `\apEVALm`.

apnum.tex

```
55: \def\apEVALn#1{\apTESTdigit#1%
56:   \iftrue \ifx E#1\afterassignment\apEVALm\expandafter\expandafter\expandafter\apE
57:   \else\edef\tmpb{\tmpb#1}\expandafter\expandafter\expandafter\apEVALn\fi
58:   \else \expandafter\apEVALo\expandafter#1\fi
59: }
```

The reading an operator is done by the `\apEVALo` macro. This is more simple because the operator is only one token. Depending on this token the macro `\apEVALp` $\langle operator \rangle \langle priority \rangle$ pushes to the stack (by the macro `\apEVALpush`) the value from `\tmpb`, the $\langle operator \rangle$ and the priority increased by `\apnumA` (level of brackets).

If there is a problem (level of brackets less than zero, level of brackets not equal to zero at the end of the expression, unknown operator) we print an error using `\apEVALerror` macro.

The `\apNext` is set to `\apEVALb`, i. e. scanner returns back to the state of reading the operand. But exceptions exist: if the `)` is found then priority is decreased and the macro `\apEVALo` is executed again. If the end of the $\langle expression \rangle$ is found then the loop is ended by `\let\apNext=\relax`.

apnum.tex

```
60: \def\apEVALo#1{\let\apNext=\apEVALb
61:   \ifx+#1\apEVALp \apPLUS 1\fi
62:   \ifx-#1\apEVALp \apMINUS 1\fi
63:   \ifx*#1\apEVALp \apMUL 2\fi
64:   \ifx/#1\apEVALp \apDIV 2\fi
65:   \ifx^#1\apEVALp \apPOWx 3\fi
66:   \ifx)#1\advance\apnumA by-4 \let\apNext=\apEVALo \let\tmpa=\relax
67:   \ifnum\apnumA<0 \apEVALerror{many brackets "}")\fi
68:   \fi
69:   \ifx\limits#1%
70:     \ifnum\apnumA>0 \apEVALerror{missing bracket "}")\let\tmpa=\relax
71:     \else \apEVALp\END 0\let\apNext=\relax \fi
72:   \fi
73:   \ifx\tmpa\relax \else \apEVALerror{unknown operator "\string#1"}\fi
74:   \apnumE=0 \apNext
75: }
76: \def\apEVALp#1#2{%
77:   \apnumB=#2 \advance\apnumB by\apnumA
78:   \toks0=\expandafter{\expandafter{\tmpb}{#1}}%
79:   \expandafter\apEVALpush\the\toks0\expandafter{\the\apnumB}% {value}{op}{priority}
80:   \let\tmpa=\relax
81: }
```

The `\apEVALstack` macro includes the stack, three items $\{\langle operand \rangle\}\{\langle operator \rangle\}\{\langle priority \rangle\}$ per level. Left part of the macro contents is the top of the stack. The stack is initialized with empty operand and operator and with priority zero. The dot here is only the “total bottom” of the stack.

apnum.tex

```
82: \def\apEVALstack{{}{\relax}{0}.}
```

`\apEVALk`: 10–11 `\apEVALm`: 11 `\apEVALn`: 10–11 `\apEVALo`: 10–11 `\apEVALp`: 10–11
`\apEVALstack`: 11–12

The macro `\apEVALpush` $\{\langle operand \rangle\}\{\langle operator \rangle\}\{\langle priority \rangle\}$ pushes its parameters to the stack and runs `\apEVALdo` $\langle whole stack \rangle$ to do the desired work on the top of the stack.

apnum.tex

```
83: \def\apEVALpush#1#2#3{% value, operator, priority
84:   \toks0={{#1}{#2}{#3}}%
85:   \expandafter\def\expandafter\apEVALstack\expandafter{\the\toks0\apEVALstack}%
86:   \expandafter\apEVALdo\apEVALstack%
87: }
```

Finally, the macro `\apEVALdo` $\{\langle vt \rangle\}\{\langle ot \rangle\}\{\langle pt \rangle\}\{\langle vp \rangle\}\{\langle op \rangle\}\{\langle pp \rangle\}\langle rest of the stack \rangle$ performs the execution described at the beginning of this section. The new operand $\langle vn \rangle$ is created as $\langle op \rangle\{\langle vp \rangle\}\{\langle vt \rangle\}$, this means `\apPLUS` $\{\langle vp \rangle\}\{\langle vt \rangle\}$ for example. The operand is not executed now, only the result is composed by the normal TeX notation. If the bottom of the stack is reached then the result is saved to the `\tmpb` macro. This macro is executed after group by the `\apEVALa` macro.

apnum.tex

```
88: \def\apEVALdo#1#2#3#4#5#6#7@{%
89:   \apnumB=#3 \ifx#2\apPOWx \advance\apnumB by1 \fi
90:   \ifnum\apnumB>#6\else
91:     \ifnum#6=0 \def\tmpb{#1}%\toks0={#1}\message{RESULT: \the\toks0}
92:     \ifnum\apnumE=1 \def\tmpb{\apPPn{#1}}\fi
93:     \else \def\apEVALstack{#7}\apEVALpush{#5{#4}{#1}}{#2}{#3}%
94:     \fi\fi
95: }
```

The macro `\apEVALerror` $\langle string \rangle$ prints an error message. We decide to be better to print only `\message`, no `\errmessage`. The `\tmpb` is prepared to create `\OUT` as ?? and the `\apNext` macro is set in order to skip the rest of the scanned $\langle expression \rangle$.

apnum.tex

```
96: \def\apEVALerror#1{\message{\noexpand\evaldef ERROR: #1.}%
97:   \def\OUT{0}\apE=0\apSIGN=0\def\apNext##1\apEND{\apEND}%
98: }
```

The auxiliary macro `\apTESTdigit` $\langle token \rangle$ `\iftrue` tests, if the given token is digit, dot or E letter.

apnum.tex

```
99: \def\apTESTdigit#1#2{%
100:   \ifx E#1\apXtrue \else
101:     \ifcat.\noexpand#1%
102:       \ifx.#1\apXtrue \else
103:         \ifnum'#1<'0 \apXfalse\else
104:           \ifnum'#1>'9 \apXfalse\else \apXtrue\fi
105:       \fi\fi
106:     \else \apXfalse
107:     \fi\fi
108:   \ifapX
109: }
```

2.3 Preparation of the Parameter

All operands of `\apPLUS`, `\apMINUS`, `\apMUL`, `\apDIV` and `\apPOW` macros are preprocessed by `\apPPa` macro. This macro solves (roughly speaking) the following tasks:

- It partially expands (by `\expandafter`) the parameter while $\langle sign \rangle$ is read.
- The $\langle sign \rangle$ is removed from parameter and the appropriate `\apSIGN` value is set.
- If the next token after $\langle sign \rangle$ is `\relax` then the rest of the parameter is executed in the group and the results `\OUT`, `\apSIGN` and `\apE` are used.
- Else the number is read and saved to the parameter.
- If the read number has the scientific notation $\langle mantissa \rangle E \langle exponent \rangle$ then only $\langle mantissa \rangle$ is saved to the parameter and `\apE` is set as $\langle exponent \rangle$. Else `\apE` is zero.

The macro `\apPPa` $\langle sequence \rangle \langle parameter \rangle$ calls `\apPPb` $\langle parameter \rangle @ \langle sequence \rangle$ and starts reading the $\langle parameter \rangle$. The result will be stored to the $\langle sequence \rangle$.

`\apEVALpush`: 11–12 `\apEVALdo`: 12 `\apEVALerror`: 11–12 `\apTESTdigit`: 10–12 `\apPPa`: 12–14
`\apPPb`: 13–14

Each token from $\langle sign \rangle$ is processed by three `\expandafters` (because there could be `\csname...\endcsname`). It means that the parameter is partially expanded when $\langle sign \rangle$ is read. The `\apPPb` macro sets the initial value of `\tmpc` and `\apSIGN` and executes the macro `\apPPc` $\langle parameter \rangle @ \langle sequence \rangle$.

apnum.tex

```

113: \def\apPPa#1#2{\expandafter\apPPb#2@#1}
114: \def\apPPb{\def\tmpc{}\apSIGN=1 \apE=0 \apXfalse \expandafter\expandafter\expandafter\apPPc}
115: \def\apPPc#1{%
116:   \ifx+#1\apPPd \fi
117:   \ifx-#1\apSIGN=-\apSIGN \apPPd \fi
118:   \ifx\relax#1\apPPe \fi
119:   \apPPg#1%
120: }
121: \def\apPPd#1\apPPg#2{\fi\expandafter\expandafter\expandafter\apPPc}

```

The `\apPPc` reads one token from $\langle sign \rangle$ and it is called recursively while there are + or - signs. If the read token is + or - then the `\apPPd` closes conditionals and executes `\apPPc` again.

If `\relax` is found then the rest of parameter is executed by the `\apPPe`. The macro ends by `\apPPf` $\langle result \rangle @$ and this macro reverses the sign if the result is negative and removes the minus sign from the front of the parameter.

apnum.tex

```

122: \def\apPPe#1\apPPg#2#3@{\fi\apXtrue
123:   \begingroup#3% execution of the parameter in the group
124:   \edef\tmpb{\apE=\the\apE\relax\noexpand\apPPf\OUT@}\expandafter\endgroup\tmpb
125: }
126: \def\apPPf#1{\ifx-#1\apSIGN=-\apSIGN \expandafter\apPPg\else\expandafter\apPPg\expandafter#1\fi}

```

The `\apPPg` $\langle parameter \rangle @$ macro is called when the $\langle sign \rangle$ was processed and removed from the input stream. The main reason of this macro is to remove trailing zeros from the left and to check, if there is the zero value written for example in the form 0000.000. When this macro is started then `\tmpc` is empty. This is a flag for removing trailing zeros. They are simply ignored before decimal point. The `\apPPg` is called again by `\apPPh` macro which removes the rest of `\apPPg` macro and closes the conditional. If the decimal point is found then next zeros are accumulated to the `\tmpc`. If the end of the parameter `@` is found and we are in the “removing zeros state” then the whole value is assumed to be zero and this is processed by `\apPPi` `@`. If another digit is found (say 2) then there are two situations: if the `\tmpc` is non-empty, then the digit is appended to the `\tmpc` and the `\apPPi` $\langle expanded tmp \rangle$ is processed (say `\apPPi .002`) followed by the rest of the parameter. Else the digit itself is stored to the `\tmpc` and it is returned back to the input stream (say `\apPPi 2`) followed by the rest of the parameter.

apnum.tex

```

127: \def\apPPg#1{%
128:   \ifx.#1\def\tmpc{.}\apPPh\fi
129:   \ifx\tmpc\empty\else\edef\tmpc{\tmpc#1}\fi
130:   \ifx0#1\apPPh\fi
131:   \ifx\tmpc\empty\edef\tmpc{#1}\fi
132:   \ifx@#1\def\tmpc{@}\apSIGN=0 \fi
133:   \expandafter\apPPi\tmpc
134: }
135: \def\apPPh#1\apPPi\tmpc{\fi\apPPg}

```

The macro `\apPPi` $\langle parameter without trailing zeros \rangle @ \langle sequence \rangle$ switches to two cases: if the execution of the parameter was processed then the `\OUT` doesn't include E notation and we can simply define $\langle sequence \rangle$ as the $\langle parameter \rangle$ by the `\apPPj` macro. This saves the copying of the (possible) long result to the input stream again.

If the executing of the parameter was not performed, then we need to test the existence of the E notation of the number by the `\apPPk` macro. We need to put the $\langle parameter \rangle$ to the input stream and to use `\apPPl` to test these cases. We need to remove unwanted E letter by the `\apPPm` macro.

apnum.tex

```

136: \def\apPPi{\ifapX \expandafter\apPPj \else \expandafter\apPPk \fi}
137: \def\apPPj#1@#2{\def#2{#1}}

```

<code>\apPPc:</code> 13	<code>\apPPd:</code> 13	<code>\apPPe:</code> 13	<code>\apPPf:</code> 13	<code>\apPPg:</code> 13	<code>\apPPh:</code> 13	<code>\apPPi:</code> 13
<code>\apPPj:</code> 13	<code>\apPPk:</code> 13–14	<code>\apPPl:</code> 14	<code>\apPPm:</code> 14			

```

138: \def\apPPk#1@#2{\ifx@#1@\apSIGN=0 \def#2{0}\else \apPPl#1E@#2\fi}
139: \def\apPPl#1E#2@#3{%
140:   \ifx@#1@\def#3{1}\else\def#3{#1}\fi
141:   \ifx@#2@\else \afterassignment\apPPm \apE=#2\fi
142: }
143: \def\apPPm E{}
```

The `\apPPn` $\langle param \rangle$ macro does the same as `\apPPa\OUT{\langle param \rangle}`, but the minus sign is returned back to the `\OUT` macro if the result is negative.

apnum.tex

```
144: \def\apPPn#1{\expandafter\apPPb#1@\OUT}
```

The `\apPPab` $\langle macro \rangle \{ \langle paramA \rangle \} \{ \langle paramB \rangle \}$ is used for parameters of all macros `\apPLUS`, `\apMUL` etc. It prepares the $\langle paramA \rangle$ to `\tmpa`, $\langle paramB \rangle$ to `\tmpb`, the sign and $\langle decimal exponent \rangle$ of $\langle paramA \rangle$ to the `\apSIGNa` and `\apEa`, the same of $\langle paramB \rangle$ to the `\apSIGNb` and `\apEb`. Finally, it executes the $\langle macro \rangle$.

apnum.tex

```

148: \def\apPPab#1#2#3{%
149:   \expandafter\apPPb#2@\tmpa \apSIGNa=\apSIGN \apEa=\apE
150:   \expandafter\apPPb#3@\tmpb \apSIGNb=\apSIGN \apEb=\apE
151:   #1%
152: }
```

The `\apPPs` $\langle macro \rangle \langle sequence \rangle \{ \langle param \rangle \}$ prepares parameters for `\apROLL`, `\apROUND` and `\apNORM` macros. It saves the $\langle param \rangle$ to the `\tmpc` macro, expands the $\langle sequence \rangle$ and runs the macro `\apPPt` $\langle macro \rangle \langle expanded sequence \rangle . @ \langle sequence \rangle$. The macro `\apPPt` reads first token from the $\langle expanded sequence \rangle$ to `#2`. If `#2` is minus sign, then `\apnumG=-1`. Else `\apnumG=1`. Finally the $\langle macro \rangle \langle expanded sequence \rangle . @ \langle sequence \rangle$ is executed (but without the minus sign in the input stream). If `#2` is zero then `\apPPu` $\langle macro \rangle \langle rest \rangle . @ \langle sequence \rangle$ is executed. If the $\langle rest \rangle$ is empty, (i. e. the parameter is simply zero) then $\langle macro \rangle$ isn't executed because there is nothing to do with zero number as a parameter of `\apROLL`, `\apROUND` or `\apNORM` macros.

apnum.tex

```

153: \def\apPPs#1#2#3{\def\tmpc{#3}\expandafter\apPPt\expandafter#1#2.@#2}
154: \def\apPPt#1#2{%
155:   \ifx-#2\apnumG=-1 \def\apNext{#1}%
156:   \else \ifx0#2\apnumG=0 \def\apNext{\apPPu#1}\else \apnumG=1 \def\apNext{#1#2}\fi\fi
157:   \apNext
158: }
159: \def\apPPu#1#2.@#3{\ifx@#2@\apnumG=0 \ifx#1\apROUNDa\def\XOUT{}\fi
160:   \else\def\apNext{\apPPt#1#2.@#3}\expandafter\apNext\fi
161: }
```

2.4 Addition and Subtraction

The significant part of the optimization in `\apPLUS`, `\apMUL`, `\apDIV` and `\apPOW` macros is the fact, that we don't treat with single decimal digits but with their quartets. This means that we are using the numeral system with the base 10000 and we calculate four decimal digits in one elementary operation. The base was chosen 10^4 because the multiplication of such numbers gives results less than 10^8 and the maximal number in the $\text{T}_{\text{E}}\text{X}$ register is about $2 \cdot 10^9$. We'll use the word "Digit" (with capitalized D) in this documentation if this means the digit in the numeral system with base 10000, i. e. one Digit is four digits. Note that for addition we can use the numeral system with the base 10^8 but we don't do it, because the auxiliary macros `\apIV*` for numeral system of the base 10^4 are already prepared.

Suppose the following example (the spaces between Digits are here only for more clarity).

```

123 4567 8901 9999      \apnumA=12 \apnumE=3 \apnumD=16
+          22.423      \apnumB=0  \apnumF=2  \apnumC=12
-----
sum in reversed order and without transmissions:
      {4230}{10021}{8901}{4567}{123}  \apnumD=-4
```

`\apPPn`: 12, 14, 47, 49 `\apPPab`: 14–15, 18, 24, 28–29, 33 `\apPPs`: 14, 18, 30–33 `\apPPt`: 14
`\apPPu`: 14


```
sum in normal order including transmissions:
123 4567 8902 0021.423
```

In the first pass, we put the number with more or equal Digits before decimal point above the second number. There are three Digits more in the example. The `\apnumC` register saves this information (multiplied by 4). The first pass creates the sum in reversed order without transmissions between Digits. It simply copies the `\apnumC/4` Digits from the first number to the result in reversed order. Then it does the sums of Digits without transmissions. The `\apnumD` is a relative position of the decimal point to the edge of the calculated number.

The second pass reads the result of the first pass, calculates transmissions and saves the result in normal order.

The first Digit of the operands cannot include four digits. The number of digits in the first Digit is saved in `\apnumE` (for first operand) and in `\apnumF` (for second one). The rule is to have the decimal point between Digits in all circumstances.

The `\apPLUS` and `\apMINUS` macros prepare parameters using `\apPPab` and execute `\apPLUSa`:

```
165: \def\apPLUS{\relax \apPPab\apPLUSa}
166: \def\apMINUS#1#2{\relax \apPPab\apPLUSa{#1}{-#2}}
```

apnum.tex

The macro `\apPLUSa` does the following work:

- It gets the operands in `\tmpa` and `\tmpb` macros using the `\apPPab`.
- If the scientific notation is used and the decimal exponents `\apEa` and `\apEb` are not equal then the decimal point of one operand have to be shifted (by the macro `\apPLUSxE` at line 168).
- The digits before decimal point are calculated for both operands by the `\apDIG` macro. The first result is saved to `\apnumA` and the second result is saved to `\apnumB`. The `\apDIG` macro removes decimal point (if exists) from the parameters (lines 169 and 170).
- The number of digits in the first Digit is calculated by `\apIVmod` for both operands. This number is saved to `\apnumE` and `\apnumF`. This number is subtracted from `\apnumA` and `\apnumB`, so these registers now includes multiply of four (lines 171 and 172).
- The `\apnumC` includes the difference of Digits before the decimal point (multiplied by four) of given operands (line 173).
- If the first operand is negative then the minus sign is inserted to the `\apPLUSxA` macro else this macro is empty. The same for the second operand and for the macro `\apPLUSxB` is done (lines 174 and 175).
- If both operands are positive, then the sign of the result `\apSIGN` is set to one. If both operands are negative, then the sign is set to -1 . But in both cases mentioned above we will do (internally) addition, so the macros `\apPLUSxA` and `\apPLUSxB` are set to empty. If one operand is negative and second positive then we will do subtraction. The `\apSIGN` register is set to zero and it will set to the right value later (lines 176 to 178).
- The macro `\apPLUSb⟨first op⟩⟨first dig⟩⟨second op⟩⟨second dig⟩⟨first Dig⟩` does the calculation of the first pass. The `⟨first op⟩` has to have more or equal Digits before decimal point than `⟨second op⟩`. This is reason why this macro is called in two variants dependent on the value `\apnumC`. The macros `\apPLUSxA` and `\apPLUSxB` (with the sign of the operands) are exchanged (by the `\apPLUSg`) if the operands are exchanged (lines 179 to 180).
- The `\apnumG` is set by the macro `\apPLUSb` to the sign of the first nonzero Digit. It is equal to zero if there are only zero Digits after first pass. The result is zero in such case and we do nothing more (line 182).
- The transmission calculation is different for addition and subtraction. If the subtraction is processed then the sign of the result is set (using the value `\apnumG`) and the `\apPLUSm` for transmissions is prepared. Else the `\apPLUSp` for transmissions is prepared as the `\apNext` macro (line 183).
- The result of the first pass is expanded in the input stream and the `\apNext` (i. e. transmissions calculation) is activated at line 184.

`\apPLUS`: 6, 9–12, 14–15, 37–39, 41–48, 50 `\apMINUS`: 6, 11–12, 15, 45, 47–48, 50 `\apPLUSa`: 15–16
`\apPLUSxA`: 15–17 `\apPLUSxB`: 15–17

- This macro adds the sign of the result too (lines 185 to 191)

apnum.tex

```

167: \def\apPLUSa{%
168:   \ifnum\apEa=\apEb \apE=\apEa \else \apPLUSxE \fi
169:   \apDIG\tmpa\relax \apnumA=\apnumD % digits before decimal point
170:   \apDIG\tmpb\relax \apnumB=\apnumD
171:   \apIVmod \apnumA \apnumE \advance\apnumA by-\apnumE % digits in the first Digit
172:   \apIVmod \apnumB \apnumF \advance\apnumB by-\apnumF
173:   \apnumC=\apnumB \advance\apnumC by-\apnumA % difference between Digits
174:   \ifnum\apSIGNa<0 \def\apPLUSxA{-}\else \def\apPLUSxA{}\fi
175:   \ifnum\apSIGNb<0 \def\apPLUSxB{-}\else \def\apPLUSxB{}\fi
176:   \apSIGN=0 % \apSIGN=0 means that we are doing subtraction
177:   \ifx\apPLUSxA\empty \ifx\apPLUSxB\empty \apSIGN=1 \fi\fi
178:   \if\apPLUSxA-\relax \if\apPLUSxB-\relax \apSIGN=-1 \def\apPLUSxA{}\def\apPLUSxB{}\fi\fi
179:   \ifnum\apnumC>0 \apPLUSg \apPLUSb \tmpb\apnumF \tmpa\apnumE \apnumB % first pass
180:   \else \apnumC=-\apnumC \apPLUSb \tmpa\apnumE \tmpb\apnumF \apnumA
181:   \fi
182:   \ifnum\apnumG=0 \def\OUT{0}\apSIGN=0 \apE=0 \else
183:     \ifnum\apSIGN=0 \apSIGN=\apnumG \let\apNext=\apPLUSm \else \let\apNext=\apPLUSp \fi
184:     \apnumX=0 \edef\OUT{\expandafter\expandafter \apNext \OUT}% second pass
185:     \ifnum\apnumD<1 % result in the form .000123
186:       \apnumZ=-\apnumD
187:       \def\tmpa{.}%
188:       \ifnum\apnumZ>0 \apADDzeros\tmpa \fi % adding dot and left zeros
189:       \edef\OUT{\ifnum\apSIGN<0-\fi\tmpa\OUT}%
190:     \else
191:       \edef\OUT{\expandafter\expandafter\apPLUSy \OUT}% removing left zeros
192:     \fi\fi
193: }

```

The $\langle first\ op \rangle$ is the first operand (which have more or equal Digits before decimal point). The $\langle first\ dig \rangle$ is the number of digits in the first Digit in the first operand. The $\langle second\ op \rangle$ is the second operand and the $\langle second\ dig \rangle$ is the number of digits in the first Digit of the second operand. The $\langle first\ Dig \rangle$ is the number of Digits before decimal point of the first operand, but without the first Digit and multiplied by 4.

parameters before this operand in order to read desired number of digits to the first Digit of this operand. The macro `\apPLUSb` saves the first operand to the input queue after `\apPLUSc` macro. It inserts the appropriate number of empty parameters (in `\tmpc`) before this operand in order to read the right number of digits in the first attempt. It appends the `\apNL` marks to the end in order to recognize the end of the input stream. These macros expands simply to zero but we can test the end of input stream by `\ifx`.

by 4) in `\apnumD` by advancing `\first DIG` by 4. It initializes `\apnumZ` to zero. If the first nonzero Digit will be found then `\apnumZ` will be set to this Digit in the `\apPLUSc` macro.

apnum.tex

```

194: \def\apPLUSb#1#2#3#4#5{%
195:   \edef\tmpd{\ifcase#4\or{}{}\or{}{}\or{}{}\fi}%
196:   \edef\tmpc{\ifcase#2\or{}{}\or{}{}\or{}{}\fi}%
197:   \let\apNext=\apPLUSc \apnumD=#5\advance\apnumD by4 \apnumG=0 \apnumZ=0 \def\OUT{}%
198:   \expandafter\expandafter\expandafter\apPLUSc\expandafter\tmpc#1\apNL\apNL\apNL\apNL%
199: }

```

the `\apnumY`. Then it calls the `\apPLUSE`, which reads (if it is allowed, i. e. if `\apnumC<=0`) one digit from second operand `\tmpd` by the `\apIVread` macro. Then it does the addition of these digits and saves the result into the `\OUT` macro in reverse order.

advancing is performed. This means that we are doing addition or subtraction here.

If the first nonzero Digit is reached, then the macro `\apPLUSH` sets the sign of the result to the `\apnumG` and (maybe) exchanges the `\apPLUSxA` and `\apPLUSxB` macros (by the `\apPLUSg` macro) in order to the internal result of the subtraction will be always non-negative.

If the end of input stream is reached, then `\apNext` (used at line 212) is reset from its original value `\apPLUSc` to the `\apPLUSd` where the `\apnumY` is simply set to zero. The reading from input stream is finished. This occurs when there are more Digits after decimal point in the second operand than in the first one. If the end of input stream is reached and the `\tmpd` macro is empty (all data from second operand was read too) then the `\apPLUSf` macro removes the rest of input stream and the first pass of the calculation is done.

```

200: \def\apPLUSc#1#2#3#4{\apnumY=\apPLUSxA#1#2#3#4\relax
201:   \ifx\apNL#4\let\apNext=\apPLUSd\fi
202:   \ifx\apNL#1\relax \ifx\tmpd\empty \expandafter\expandafter\expandafter\apPLUSf \fi\fi
203:   \apPLUSe
204: }
205: \def\apPLUSd{\apnumY=0 \ifx\tmpd\empty \expandafter\apPLUSf \else\expandafter \apPLUSe\fi}
206: \def\apPLUSe{%
207:   \ifnum\apnumC>0 \advance\apnumC by-4
208:   \else \apIVread\tmpd \advance\apnumY by\apPLUSxB\apnumX \fi
209:   \ifnum\apnumZ=0 \apPLUSh \fi
210:   \edef\OUT{{\the\apnumY}\OUT}%
211:   \advance\apnumD by-4
212:   \apNext
213: }
214: \def\apPLUSf#1@{}
215: \def\apPLUSg{\let\tmpc=\apPLUSxA \let\apPLUSxA=\apPLUSxB \let\apPLUSxB=\tmpc}
216: \def\apPLUSh{\apnumZ=\apnumY

```

apnum.tex

Why there is a complication about reading one parameter from input stream but second one from the macro `\tmpd`? This is more faster than to save both parameters to the macros and using `\apIVread` for both because the `\apIVread` must redefine its parameter. You can examine that this parameter is very long.

The `\apPLUSm` `<data>@` macro does transmissions calculation when subtracting. The `<data>` from first pass is expanded in the input stream. The `\apPLUSm` macro reads repeatedly one Digit from the `<data>` until the stop mark is reached. The Digits are in the range -9999 to 9999 . If the Digit is negative then we need to add 10000 and set the transmission value `\apnumX` to one, else `\apnumX` is zero. When the next Digit is processed then the calculated transmission value is subtracted. The macro `\apPLUSw` writes the result for each Digit `\apnumA` in the normal (human readable) order.

```

219: \def\apPLUSm#1{%
220:   \ifx@#1\else
221:     \apnumA=#1 \advance\apnumA by-\apnumX
222:     \ifnum\apnumA<0 \advance\apnumA by\apIVbase \apnumX=1 \else \apnumX=0 \fi
223:     \apPLUSw
224:     \expandafter\apPLUSm
225:   \fi
226: }

```

apnum.tex

The `\apPLUSp` `<data>@` macro does transmissions calculation when addition is processed. It is very similar to `\apPLUSm`, but Digits are in the range 0 to 19998 . If the Digit value is greater than 9999 then we need to subtract 10000 and set the transmission value `\apnumX` to one, else `\apnumX` is zero.

```

227: \def\apPLUSp#1{%
228:   \ifx@#1\ifnum\apnumX>0 \apnumA=1 \apPLUSw \fi %.5+.5=.1 bug fixed
229:   \else
230:     \apnumA=\apnumX \advance\apnumA by#1
231:     \ifnum\apnumA<\apIVbase \apnumX=0 \else \apnumX=1 \advance\apnumA by-\apIVbase \fi
232:     \apPLUSw
233:     \expandafter\apPLUSp
234:   \fi
235: }

```

apnum.tex

`\apPLUSh`: 17 `\apPLUSg`: 15–17 `\apPLUSd`: 17 `\apPLUSf`: 17 `\apPLUSm`: 15–17
`\apPLUSp`: 15–17

The `\apPLUSw` writes the result with one Digit (saved in `\apnumA`) to the `\OUT` macro. The `\OUT` is initialized as empty. If it is empty (it means we are after decimal point), then we need to write all four digits by `\apIVwrite` macro (including left zeros) but we need to remove right zeros by `\apREMzerosR`. If the decimal point is reached, then it is saved to the `\OUT`. But if the previous `\OUT` is empty (it means there are no digits after decimal point or all such digits are zero) then `\def\OUT{\empty}` ensures that the `\OUT` is non-empty and the ignoring of right zeros are disabled from now.

```
236: \def\apPLUSw{%
237:   \ifnum\apnumD=0 \ifx\OUT\empty \def\OUT{\empty}\else \edef\OUT{.\OUT}\fi \fi
238:   \advance\apnumD by4
239:   \ifx\OUT\empty \edef\tmpa{\apIVwrite\apnumA}\edef\OUT{\apREMzerosR\tmpa}%
240:   \else \edef\OUT{\apIVwrite\apnumA\OUT}\fi
241: }
```

apnum.tex

The macro `\apPLUSy` (*expanded OUT*)@ removes left trailing zeros from the `\OUT` macro and saves the possible minus sign by the `\apPLUSz` macro.

```
242: \def\apPLUSy#1{\ifx0#1\expandafter\apPLUSy\else \expandafter\apPLUSz\expandafter#1\fi}
243: \def\apPLUSz#1@{\edef\OUT{\ifnum\apSIGN<0-\fi#1}}
```

apnum.tex

The macro `\apPLUSxE` uses the `\apROLLa` in order to shift the decimal point of the operand. We need to set the same decimal exponent in scientific notation before the addition or subtraction is processed.

```
244: \def\apPLUSxE{%
245:   \apnumE=\apEa \advance\apnumE by-\apEb
246:   \ifnum\apEa>\apEb \apPPs\apROLLa\tmpb{-\apnumE}\apE=\apEa
247:   \else \apPPs\apROLLa\tmpa{\apnumE}\apE=\apEb \fi
248: }
```

apnum.tex

2.5 Multiplication

Suppose the following multiplication example: $1234 \times 567 = 699678$.

Normal format:		Mirrored format:
1 2 3 4 *		4 3 2 1 *
5 6 7		7 6 5
-----		-----
*7: 7 14 21 28		*7: 28 21 14 7
*6: 6 12 18 24		*6: 24 18 12 6
*5: 5 10 15 20		*5: 20 15 10 5
-----		-----
6 9 9 6 7 8		8 7 6 9 9 6

This example is in numeral system of base 10 only for simplification, the macros work really with base 10000. Because we have to do the transmissions between Digit positions from right to left in the normal format and because it is more natural for \TeX to put the data into the input stream and read it sequentially from left to right, we use the mirrored format in our macros.

The macro `\apMUL` prepares parameters using `\apPPab` and executes `\apMULa`

```
252: \def\apMUL{\relax \apPPab\apMULa}
```

apnum.tex

The macro `\apMULa` does the following:

- It gets the parameters in `\tmpa` and `\tmpb` preprocessed using the `\apPPab` macro.
- It evaluates the exponent of ten `\apE` which is usable when the scientific notation of numbers is used (line 254).
- It calculates `\apSIGN` of the result (line 255).
- If `\apSIGN=0` then the result is zero and we will do nothing more (line 256).
- The decimal point is removed from the parameters by `\apDIG{param}\register`. The `\apnumD` includes the number of digits before decimal point (after the `\apDIG` is used) and the `\register`

`\apPLUSw`: 17–18 `\apPLUSy`: 16, 18 `\apPLUSz`: 18 `\apPLUSxE`: 15–16, 18 `\apMUL`: 6, 9–12, 14,
18, 26, 36, 38, 40, 42, 44–45, 47–48, 50 `\apMULa`: 18–19, 28

includes the number of digits in the rest. The `\apnumA` or `\apnumB` includes total number of digits in the parameters `\tmpa` or `\tmpb` respectively. The `\apnumD` is re-calculated: it saves the number of digits after decimal point in the result (lines 257 to 259).

- Let A is the number of total digits in the $\langle param \rangle$ and let $F = A \bmod 4$, but if $F = 0$ then reassign it to $F = 4$. Then F means the number of digits in the first Digit. This calculation is done by `\apIVmod{A}{F}` macro. All another Digits will have four digits. The `\apMULb{param}@@@@` is able to read four digits, next four digits etc. We need to insert appropriate number of empty parameters before the $\langle param \rangle$. For example `\apMULb{}{}{}{}{param}@@@@` reads first only one digit from $\langle param \rangle$, next four digits etc. The appropriate number of empty parameters are prepared in the `\tmpc` macro (lines 260 to 261).
- The `\apMULb` reads the $\langle paramA \rangle$ (all Digits) and prepares the `\OUT` macro in the special interleaved format (described below). The format is finished by `*`. in the line 263.
- Analogical work is done with the second parameter $\langle paramB \rangle$. But this parameter is processed by `\apMULc`, which reads Digits of the parameter and inserts them to the `\tmpa` in the reversed order (lines 264 to 266).
- The main calculation is done by `\apMULd{paramB}@`, which reads Digits from $\langle paramB \rangle$ (in reversed order) and does multiplication of the $\langle paramA \rangle$ (saved in the `\OUT`) by these Digits (line 267).
- The `\apMULg` macro converts the result `\OUT` to the human readable form (line 268).
- The possible minus sign and the trailing zeros of results of the type `.00123` is prepared by `\apADDzeros\tmpa` to the `\tmpa` macro. This macro is appended to the result in the `\OUT` macro (lines 269 to 271).

apnum.tex

```

253: \def\apMULa{%
254:   \apE=\apEa \advance\apE by\apEb
255:   \apSIGN=\apSIGNa \multiply\apSIGN by\apSIGNb
256:   \ifnum\apSIGN=0 \def\OUT{0}\apE=0 \else
257:     \apDIG\tmpa\apnumA \apnumX=\apnumA \advance\apnumA by\apnumD
258:     \apDIG\tmpb\apnumB \advance\apnumX by\apnumB \advance\apnumB by\apnumD
259:     \apnumD=\apnumX % \apnumD = the number of digits after decimal point in the result
260:     \apIVmod \apnumA \apnumF % \apnumF = digits in the first Digit of \tmpa
261:     \edef\tmpc{\ifcase\apnumF\or{}\fi}\or{}\fi}\def\OUT{}%
262:     \expandafter\expandafter\expandafter \apMULb \expandafter \tmpc \tmpa @@@@%
263:     \edef\OUT{*.\OUT}%
264:     \apIVmod \apnumB \apnumF % \apnumF = digits in the first Digit of \tmpb
265:     \edef\tmpc{\ifcase\apnumF\or{}\fi}\or{}\fi}\def\tmpa{}%
266:     \expandafter\expandafter\expandafter \apMULc \expandafter \tmpc \tmpb @@@@%
267:     \expandafter\apMULD \tmpa@%
268:     \expandafter\apMULg \OUT
269:     \edef\tmpa{\ifnum\apSIGN<0-\fi}%
270:     \ifnum\apnum>0 \apnumZ=\apnumD \edef\tmpa{\tmpa.}\apADDzeros\tmpa \fi
271:     \ifx\tmpa\empty \else \edef\OUT{\tmpa\OUT}\fi
272:   \fi
273: }

```

We need to read the two data streams when the multiplication of the $\langle paramA \rangle$ by one Digit from $\langle paramB \rangle$ is performed and the partial sum is actualized. First: the digits of the $\langle paramA \rangle$ and second: the partial sum. We can save these streams to two macros and read one piece of information from such macros at each step, but this is not effective because the whole stream has to be read and redefined at each step. For \TeX is more natural to put one data stream to the input queue and to read pieces of information thereof. Thus we interleave both data streams into one `\OUT` in such a way that one element of data from first stream is followed by one element from second stream and it is followed by second element from first stream etc. Suppose that we are at the end of $i - th$ line of the multiplication scheme where we have the partial sums s_n, s_{n-1}, \dots, s_0 and the Digits of $\langle paramA \rangle$ are d_k, d_{k-1}, \dots, d_0 . The zero index belongs to the most right position in the mirrored format. The data will be prepared in the form:

$$\cdot \{s_n\} \{s_{(n-1)}\} \dots \{s_{(k+1)}\} * \{s_k\} \{d_{(k-1)}\} \dots \{s_1\} \{d_1\} \{s_0\} \{d_0\} *$$

For our example (there is a simplification: numeral system of base 10 is used and no transmissions are processed), after second line (multiplication by 6 and calculation of partial sums) we have in `\OUT`:


```
. {28} * {45} {4} {32} {3} {19} {2} {6} {1} *
```

and we need to create the following line during calculation of next line of multiplication scheme:

```
. {28} {45} * {5*4+32} {4} {5*3+19} {3} {5*2+6} {2} {5*1} {1} *
```

This special format of data includes two parts. After the starting dot, there is a sequence of sums which are definitely calculated. This sequence is ended by first `*` mark. The last definitely calculated sum follows this mark. Then the partial sums with the Digits of $\langle paramA \rangle$ are interleaved and the data are finalized by second `*`. If the calculation processes the second part of the data then the general task is to read two data elements (partial sum and the Digit) and to write two data elements (the new partial sum and the previous Digit). The line calculation starts by copying of the first part of data until the first `*` and appending the first data element after `*`. Then the `*` is written and the middle processing described above is started.

The macro `\apMULb` $\langle paramA \rangle$ prepares the special format of the macro `\OUT` described above where the partial sums are zero. It means:

```
* . {d_k} 0 {d_{k-1}} 0 ... 0 {d_0} *
```

where d_i are Digits of $\langle paramA \rangle$ in reversed order.

The first “sum” is only dot. It will be moved before `*` during the first line processing. Why there is such special “pseudo-sum”? The `\apMULe` with the parameter delimited by the first `*` is used in the context `\apMULe.{\sum}*` during the third line processing and the dot here protects from removing the braces around the first real sum.

```
274: \def\apMULb#1#2#3#4{\ifx#4\else
275:   \ifx\OUT\empty \edef\OUT{{#1#2#3#4}*}\else\edef\OUT{{#1#2#3#4}0\OUT}\fi
276:   \expandafter\apMULb\fi
277: }
```

apnum.tex

The macro `\apMULc` $\langle paramB \rangle$ reads Digits from $\langle paramB \rangle$ and saves them in reversed order into `\tmpa`. Each Digit is enclosed by \TeX braces `{}`.

```
278: \def\apMULc#1#2#3#4{\ifx#4\else \edef\tmpa{{#1#2#3#4}\tmpa}\expandafter\apMULc\fi}
```

apnum.tex

The macro `\apMULd` $\langle paramB \rangle$ reads the Digits from $\langle paramB \rangle$ (in reversed order), uses them as a coefficient for multiplication stored in `\tmpnumA` and processes the `\apMULe` $\langle special data format \rangle$ for each such coefficient. This corresponds with one line in the multiplication scheme.

```
279: \def\apMULd#1{\ifx#1\else
280:   \apnumA=#1 \expandafter\apMULe \OUT
281:   \expandafter\apMULd
282:   \fi
283: }
```

apnum.tex

The macro `\apMULe` $\langle special data format \rangle$ copies the first part of data format to the `\OUT`, copies the next element after first `*`, appends `*` and does the calculation by `\apMULf`. The `\apMULf` is recursively called. It reads the Digit to `#1` and the partial sum to the `#2` and writes `{\apnumA*#1+#2}{#1}` to the `\OUT` (lines 295 to 299). If we are at the end of data, then `#2` is `*` and we write the `{\apnumA*#1}{#1}` followed by ending `*` to the `\OUT` (lines 288 to 290).

```
284: \def\apMULe#1*#2{\apnumX=0 \def\OUT{{#1}{#2}*}\def\apOUT1{}\apnum0=1 \apnumL=0 \apMULf}
285: \def\apMULf#1#2{%
286:   \advance\apnum0 by-1 \ifnum\apnum0=0 \apOUTx \fi
287:   \apnumB=#1 \multiply\apnumB by\apnumA \advance\apnumB by\apnumX
288:   \ifx*#2%
289:     \ifnum\apnumB<\apIVbase
290:       \edef\OUT{\OUT\expandafter\apOUTs\apOUT1.,\ifnum\the\apnumB#1=0 \else{\the\apnumB}{#1}\fi*}%
291:     \else \apIVtrans
292:       \expandafter \edef\csname apOUT:\apOUTn\endcsname
293:         {\csname apOUT:\apOUTn\endcsname{\the\apnumB}{#1}}%
294:       \apMULf0*\fi
```

apnum.tex

`\apMULb`: 19–20, 28–29 `\apMULc`: 19–20 `\apMULd`: 19–20, 29 `\apMULe`: 20–21, 30
`\apMULf`: 20–21, 30


```

295: \else \advance\apnumB by#2
296: \ifnum\apnumB<\apIVbase \apnumX=0 \else \apIVtrans \fi
297: \expandafter
298: \edef\csname apOUT:\apOUTn\endcsname{\csname apOUT:\apOUTn\endcsname{\the\apnumB}\#1}}%
299: \expandafter\apMULf \fi
300: }

```

There are several complications in the algorithm described above.

- The result isn't saved directly to the `\OUT` macro, but partially into the macros `\apOUT:<num>`, as described in the section 2.9 where the `\apOUTx` macro is defined.
- The transmissions between Digit positions are calculated. First, the transmission value `\apnumX` is set to zero in the `\apMULe`. Then this value is subtracted from the calculated value `\apnumB` and the new transmission is calculated using the `\apIVtrans` macro if `\apnumB \geq 10000`. This macro modifies `\apnumB` in order it is right Digit in our numeral system.
- If the last digit has nonzero transmission, then the calculation isn't finished, but the new pair `{<transmission>}{0}` is added to the `\OUT`. This is done by recursively call of `\apMULf` at line 294.
- The another situation can be occurred: the last pair has both values zeros. Then we needn't to write this zero to the output. This is solved by the test `\ifnum\the\apnumB#1=0` at line 290.

The macro `\apMULg <special data format>@` removes the first dot (it is the #1 parameter) and prepares the `\OUT` to writing the result in reverse order, i. e. in human readable form. The next work is done by `\apMULh` and `\apMULi` macros. The `\apMULh` repeatedly reads the first part of the special data format (Digits of the result are here) until the first * is found. The output is stored by `\apMULo<digits>{\<data>}` macro. If the first * is found then the `\apMULi` macro repeatedly reads the triple `{<Digit of result>}{<Digit of A>}{<next Digit of result>}` and saves the first element in full (four-digits) form by the `\apIVwrite` if the third element isn't the stop-mark *. Else the last Digit (first Digit in the human readable form) is saved by `\the`, because we needn't the trailing zeros here. The third element is put back to the input stream but it is ignored by `\apMULj` macro when the process is finished.

```

301: \def\apMULg#1{\def\OUT{\apMULh}
302: \def\apMULh#1{\ifx*#1\expandafter\apMULi
303: \else \apnumA=#1 \apMULo4{\apIVwrite\apnumA}%
304: \expandafter\apMULh
305: \fi
306: }
307: \def\apMULi#1#2#3{\apnumA=#1
308: \ifx*#3\apMULo{\apNUMdigits\tmpa}{\the\apnumA}\expandafter\apMULj
309: \else \apMULo4{\apIVwrite\apnumA}\expandafter\apMULi
310: \fi{#3}%
311: }
312: \def\apMULj#1{}

```

apnum.tex

The `\apMULo <digits>{\<data>}` appends `<data>` to the `\OUT` macro. The number of digits after decimal point `\apnumD` is decreased by the number of actually printed digits `<digits>`. If the decimal point is to be printed into `<data>` then it is performed by the `\apMULt` macro.

```

313: \def\apMULo#1#2{\edef\tmpa{#2}%
314: \advance\apnumD by-#1
315: \ifnum\apnumD<1 \ifnum\apnumD>-4 \apMULt\fi\fi
316: \edef\OUT{\tmpa\OUT}%
317: }
318: \def\apMULt{\edef\tmpa{\apIVdot{-\apnumD}\tmpa}\edef\tmpa{\tmpa}}

```

apnum.tex

2.6 Division

Suppose the following example:

<paramA> : <paramB>	<output>
12345:678 = [12:6=2]	2 (2->1)
2*678 -1356	

```

\apMULg: 19, 21    \apMULh: 21    \apMULi: 21    \apMULj: 21    \apMULo: 21
\apMULt: 21

```

```

-1215 <0 correction! 1
12345
1*678 -678
      5565      [55:6=8] 9 (9->8)
9*678 -6102
      -537 <0 correction! 8
      5565
8*678 -5424
      1410      [14:6=2] 2
2*678 -1356
      0540      [05:6=0] 0
0*678 -0
      5400      [54:6=8] 9 (2x correction: 9->8, 8->7)
      ...
12345:678 = 182079...

```

We implement the division similar like pupils do it in the school (only the numeral system with base 10000 instead 10 is actually used, but we keep with base 10 in our illustrations). At each step of the operation, we get first two Digits from the dividend or remainder (called partial dividend or remainder) and do divide it by the first nonzero Digit of the divisor (called partial divisor). Unfortunately, the resulted Digit cannot be the definitive value of the result. We are able to find out this after the whole divisor is multiplied by resulted Digit and compared with the whole remainder. We cannot do this test immediately but only after a lot of following operations (imagine that the remainder and divisor have a huge number of Digits).

We need to subtract the remainder by the multiple of the divisor at each step. This means that we need to calculate the transmissions from the Digit position to the next Digit position from right to left (in the scheme illustrated above). Thus we need to reverse the order of Digits in the remainder and divisor. We do this reversion only once at the preparation state of the division and we interleave the data from the divisor and the dividend (the dividend will be replaced by the remainder, next by next remainder etc.).

The number of Digits of the dividend can be much greater than the number of Digits of the divisor. We need to calculate only with the first part of dividend/remainder in such case. We need to get only one new Digit from the rest of dividend at each calculation step. The illustration follows:

```

...used dividend.. | ... rest of dividend ... | .... divisor ....
1234567890123456789 7890123456789012345678901234 : 1231231231231231
xxxxxxxxxxxxxxxxxxxx 7 <- calculated remainder
xxxxxxxxxxxxxxxxxxxx x8 <- new calculated remainder
xxxxxxxxxxxxxxxxxxxx xx9 <- new calculated remainder etc.

```

We'll interleave only the “used dividend” part with the divisor at the preparation state. We'll put the “rest of dividend” to the input stream in the normal order. The macros do the iteration over calculation steps and they can read only one new Digit from this input stream if they need it. This approach needs no manipulation with the (potentially long) “rest of the dividend” at each step. If the divisor has only one Digit (or comparable small Digits) then the algorithm has only linear complexity with respect to the number of Digits in the dividend.

The numeral system with the base 10000 brings a little problem: we are simply able to calculate the number of digits which are multiple of four. But user typically wishes another number of calculated decimal digits. We cannot simply strip the trailing digits after calculation because the user needs to read the right remainder. This is a reason why we calculate the number of digits for the first Digit of the result. All another calculated Digits will have four digits. We need to prepare the first “partial dividend” in order to the F digits will be calculated first. How to do it? Suppose the following illustration of the first two Digits in the “partial remainder” and “partial divisor”:

```

0000 7777 : 1111 = 7    .. one digit in the result
0007 7778 : 1111 = 70   .. two digits in the result
0077 7788 : 1111 = 700  .. three digits in the result
0777 7888 : 1111 = 7000 .. four digits in the result

```

7777 8888 : 1111 = ????. .. not possible in the numeral system of base 10000

We need to read $F - 1$ digits to the first Digit and four digits to the second Digit of the “partial dividend”. But this is true only if the dividend is “comparably greater or equal to” divisor. The word “comparably greater” means that we ignore signs and the decimal point in compared numbers and we assume the decimal points in the front of both numbers just before the first nonzero digit. It is obvious that if the dividend is “comparably less” than divisor then we need to read F digits to the first Digit.

The macro `\apDIV` runs `\apDIVa` macro which uses the `\tmpa` (dividend) and `\tmpb` (divisor) macros and does the following work:

- If the divisor `\tmpb` is equal to zero, print error and do nothing more (line 324).
- The `\apSIGN` of the result is calculated (line 325).
- If the dividend `\tmpa` is equal to zero, then `\OUT` and `\XOUT` are zeros and do nothing more (line 326).
- Calculate the exponent of ten `\apE` when scientific notation is used (Line 326).
- The number of digits before point are counted by `\apDIG` macro for both parameters. The difference is saved to `\apnumD` and this is the number of digits before decimal point in the result (the exception is mentioned later). The `\apDIG` macro removes the decimal point and (possible) left zeros from its parameter and saves the result to the `\apnumD` register (lines 328 to 330).
- The macro `\apDIVcomp⟨paramA⟩⟨paramB⟩` determines if the `⟨paramA⟩` is “comparably greater or equal” to `⟨paramB⟩`. The result is stored in the boolean value `apX`. We can ask to this by the `\ifapX⟨true⟩\else⟨false⟩\fi` construction (line 331).
- If the dividend is “comparably greater or equal” to the divisor, then the position of decimal point in the result `\apnumD` has to be shifted by one to the right. The same is completed with `\apnumH` where the position of decimal point of the remainder will be stored (line 332).
- The number of desired digits in the result `\apnumC` is calculated (lines 333 to 339).
- If the number of desired digits is zero or less than zero then do nothing more (line 339).
- Finish the calculation of the position of decimal point in the remainder `\apnumH` (line 332).
- Calculate the number of digits in the first Digit `\apnumF` (line 343).
- Read first four digits of the divisor by the macro `\apIVread⟨sequence⟩`. Note that this macro puts trailing zeros to the right if the data stream `⟨param⟩` is shorter than four digits. If it is empty then the macro returns zero. The returned value is saved in `\apnumX` and the `⟨sequence⟩` is redefined by new value of the `⟨param⟩` where the read digits are removed (line 344).
- We need to read only `\apnumF` (or `\apnumF - 1`) digits from the `\tmpa`. This is done by the `\apIVreadX` macro at line 346. The second Digit of the “partial dividend” includes four digits and it is read by `\apIVread` macro at line 348.
- The “partial dividend” is saved to the `\apDIVxA` macro and the “partial divisor” is stored to the `\apDIVxB` macro. Note, that the second Digit of the “partial dividend” isn’t expanded by simply `\the`, because when `\apnumX=11` and `\apnumA=2222` (for example), then we need to save 22220011. These trailing zeros from left are written by the `\apIVwrite` macro (lines 349 to 350).
- The `\XOUT` macro for the currently computed remainder is initialized. The special interleaved data format of the remainder `\XOUT` is described below (line 351).
- The `\OUT` macro is initialized. The `\OUT` is generated as literal macro. First possible `⟨sign⟩`, then digits. If the number of effective digits before decimal point `\apnumD` is negative, the result will be in the form .000123 and we need to add the zeros by the `\apADDzeros` macro (lines 352 to 353).
- The registers for main loop are initialized. The `\apnumE` signalizes that the remainder of the partial step is zero and we can stop the calculation. The `\apnumZ` will include the Digit from the input stream where the “rest of dividend” will be stored (line 353).
- The main calculation loop is processed by the `\apDIVg` macro (line 355).
- If the division process stops before the position of the decimal point in the result (because there is zero remainder, for example) then we need to add the rest of zeros by `\apADDzeros` macro. This is actual for the results of the type 1230000 (line 356).
- If the remainder isn’t equal to zero, we need to extract the digits of the remainder from the special data format to the human readable form. This is done by the `\apDIVv` macro. The decimal point is inserted to the remainder by the `\apROLLa` macro (lines 358 to 359).

`\apDIV`: 6, 11–12, 14, 24, 36–50 `\apDIVa`: 24, 29

```

322: \def\apDIV{\relax \apPPab\apDIVa}
323: \def\apDIVa{%
324:   \ifnum\apSIGNb=0 \apERR{Dividing by zero}\else
325:     \apSIGN=\apSIGNa \multiply\apSIGN by\apSIGNb
326:     \ifnum\apSIGNa=0 \def\OUT{0}\def\XOUT{0}\apE=0 \apSIGN=0 \else
327:       \apE=\apEa \advance\apE by-\apEb
328:       \apDIG\tmpb\relax \apnumB=\apnumD
329:       \apDIG\tmpa\relax \apnumH=\apnumD
330:       \advance\apnumD by-\apnumB % \apnumD = num. of digits before decimal point in the result
331:       \apDIVcomp\tmpa\tmpb % apXtrue <=> A>=B, i.e 1 digit from A/B
332:       \ifapX \advance\apnumD by1 \advance\apnumH by1 \fi
333:       \apnumC=\apTOT
334:       \ifnum\apTOT<0 \apnumC=-\apnumC
335:         \ifnum\apnumD>\apnumC \apnumC=\apnumD \fi
336:       \fi
337:       \ifnum\apTOT=0 \apnumC=\apFRAC \advance\apnumC by\apnumD
338:       \else \apnumX=\apFRAC \advance\apnumX by\apnumD
339:         \ifnum\apnumC>\apnumX \apnumC=\apnumX \fi
340:       \fi
341:       \ifnum\apnumC>0 % \apnumC = the number of digits in the result
342:         \advance\apnumH by-\apnumC % \apnumH = the position of decimal point in the remainder
343:         \apIVmod \apnumC \apnumF % \apnumF = the number of digits in the first Digit
344:         \apIVread\tmpb \apnumB=\apnumX % \apnumB = partial divisor
345:         \apnumX=\apnumF \ifapX \advance\apnumX by-1 \fi
346:         \apIVreadX\apnumX\tmpa
347:         \apnumA=\apnumX % \apnumA = first Digit of the partial dividend
348:         \apIVread\tmpa % \apnumX = second Digit of the partial dividend
349:         \edef\apDIVxA{\the\apnumA\apIVwrite\apnumX}% first partial dividend
350:         \edef\apDIVxB{\the\apnumB}% partial divisor
351:         \edef\XOUT{\the\apDIVxB\the\apnumX}\@{\the\apnumA}}% the \XOUT is initialized
352:         \edef\OUT{\ifnum\apSIGN<0-\fi}%
353:         \ifnum\apnumD<0 \edef\OUT{\OUT.}\apnumZ=-\apnumD \apADDzeros\OUT \fi
354:         \apnumE=1 \apnumZ=0
355:         \let\apNext=\apDIVg \apNext % <--- the main calculation loop is here
356:         \ifnum\apnumD>0 \apnumZ=\apnumD \apADDzeros\OUT \fi
357:         \ifnum\apnumE=0 \def\XOUT{0}\else % extracting remainder from \XOUT
358:           \edef\XOUT{\expandafter}\expandafter\apDIVv\XOUT
359:           \def\tmpc{\apnumH}\apnumG=\apSIGNa \expandafter\apROLLa\XOUT.@\XOUT
360:         \fi
361:       \else
362:         \def\OUT{0}\def\XOUT{0}\apE=0 \apSIGN=0
363:       \fi\fi\fi
364: }

```

The macro `\apDIVcomp` $\langle paramA \rangle \langle paramB \rangle$ provides the test if the $\langle paramA \rangle$ is “comparably greater or equal” to $\langle paramB \rangle$. Imagine the following examples:

```

123456789 : 123456789 = 1
123456788 : 123456789 = .99999999189999992628

```

The example shows that the last digit in the operands can be important for the first digit in the result. This means that we need to compare whole operands but we can stop the comparison when the first difference in the digits is found. This is lexicographic ordering. Because we don’t assume the existence of eTeX (or another extensions), we need to do this comparison by macros. We set the $\langle paramA \rangle$ and $\langle paramB \rangle$ to the `\tmpc` and `\tmpd` respectively. The trailing `\apNLs` are appended. The macro `\apDIVcompA` reads first 8 digits from first parameter and the macros `\apDIVcompB` reads first 8 digits from second parameter and does the comparison. If the numbers are equal then the loop is processed again.

```

365: \def\apDIVcomp#1#2{%
366:   \expandafter\def\expandafter\tmpc\expandafter{#1\apNL\apNL\apNL\apNL\apNL\apNL\apNL\apNL}%
367:   \expandafter\def\expandafter\tmpd\expandafter{#2\apNL\apNL\apNL\apNL\apNL\apNL\apNL\apNL}%
368:   \def\apNext{\expandafter\expandafter\expandafter\apDIVcompA\expandafter\tmpc\tmpd}%
369:   \apXtrue \apNext

```

`\apDIVcomp`: 23–24 `\apDIVcompA`: 24–25 `\apDIVcompB`: 25

```

370: }
371: \def\apDIVcompA#1#2#3#4#5#6#7#8#9@{%
372:   \ifx#8\apNL \def\tmpc{0000000\apNL@}\else\def\tmpc{#9@}\fi
373:   \apnumX=#1#2#3#4#5#6#7#8\relax
374:   \apDIVcompB
375: }
376: \def\apDIVcompB#1#2#3#4#5#6#7#8#9@{%
377:   \ifnum\apnumX<#1#2#3#4#5#6#7#8 \let\apNext=\relax \apXfalse \else
378:   \ifnum\apnumX>#1#2#3#4#5#6#7#8 \let\apNext=\relax \apXtrue
379:   \fi\fi
380:   \ifx\apNext\relax\else
381:     \ifx#8\apNL \def\tmpd{0000000\apNL@}\ifx\tmpc\tmpd\let\apNext=\relax\fi
382:     \else\def\tmpd{#9@}\fi
383:     \fi
384:     \apNext
385: }

```

The format of interleaved data with divisor and remainder is described here. Suppose this partial step of the division process:

R0	R1	R2	R3	...	Rn	:	d1	d2	d3	...	dn	=	...A...
@	-A*d1	-A*d2	-A*d3	...	-A*dn								[R0 R1 : d1 = A]
0	N0	N1	N2	...	N(n-1) Nn								

The R_k are Digits of the remainder, d_k are Digits of the divisor. The A is calculated Digit in this step. The calculation of the Digits of the new remainder is hinted here. We need to do this from right to left because of the transmissions. This implies, that the interleaved format of `\XOUT` is in the reverse order and looks like

dn	Rn	...	d3	R3	d2	R2	d1	R1	@	R0
----	----	-----	----	----	----	----	----	----	---	----

for example for $\langle paramA \rangle = 1234567893$, $\langle paramB \rangle = 454502$ (in the human readable form) the `\XOUT` should be `{200}{9300}{4545}{5678}@{1234}` (in the special format). The Digits are separated by \TeX braces `{}`. The resulted digit for this step is $A = 12345678/1415 = 2716$.

The calculation of the new remainder takes d_k , R_k , d_{k-1} for each k from n to 0 and creates the Digit of the new remainder $N_{k-1} = R_k - A \cdot d_k$ (roughly speaking, actually it calculates transmissions too) and adds the new couple d_{k-1} N_{k-1} to the new version of `\XOUT` macro. The zero for N_{-1} should be reached. If it is not completed then a correction of the type $A := A - 1$ have to be done and the calculation of this step is processed again.

The result in the new `\XOUT` should be (after one step is done):

dn	Nn	...	d3	N3	d2	N2	d1	N1	@	N0
----	----	-----	----	----	----	----	----	----	---	----

where N_n is taken from the “rest of the dividend” from the input stream.

The initialization for the main loop is done by `\apDIVg` macro. It reads the Digits from `\tmpa` (dividend) and `\tmpb` macros (using `\apIVread`) and appends them to the `\XOUT` in described data format. This initialization is finished when the `\tmpb` is empty. If the `\tmpa` is not empty in such case, we put it to the input stream using `\expandafter\apDIVh\tmpa` followed by four `\apNLs` (which simply expands zero digit) followed by stop-mark. The `\apDIVh` reads one Digit from input stream. Else we put only the stop-mark to the input stream and run the `\apDIVi`. The `\apNexti` is set to the `\apDIVi`, so the macro `\apDIVh` will be skipped forever and no new Digit is read from input stream.

```

386: \def\apDIVg{%
387:   \ifx\tmpb\empty
388:     \ifx\tmpa\empty \def\apNext{\apDIVi!}\let\apNexti=\apDIVi
389:     \else \def\apNext{\expandafter\apDIVh\tmpa\apNL\apNL\apNL\apNL!}\let\apNexti=\apDIVh
390:     \fi\fi
391:     \ifx\apNext\apDIVg
392:       \apIVread\tmpa \apnumA=\apnumX
393:       \apIVread\tmpb
394:       \edef\XOUT{{\the\apnumX}{\the\apnumA}\XOUT}%
395:       \fi
396:       \apNext

```

apnum.tex

```
397: }
```

The macro `\apDIVh` reads one Digit from data stream (from the rest of the dividend) and saves it to the `\apnumZ` register. If the stop-mark is reached (this is recognized that the last digit is the `\apNL`), then `\apNexti` is set to `\apDIVi`, so the `\apDIVh` is never processed again.

apnum.tex

```
398: \def\apDIVh#1#2#3#4{\apnumZ=#1#2#3#4
399:   \ifx\apNL#4\let\apNexti=\apDIVi\fi
400:   \apDIVi
401: }
```

The macro `\apDIVi` contains the main loop for division calculation. The core of this loop is the macro call `\apDIVp<data>` which adds next digit to the `\OUT` and recalculates the remainder.

The macro `\apDIVp` decreases the `\apnumC` register (the desired digits in the output) by four, because four digits will be calculated in the next step. The loop is processed while `\apnumC` is positive. The `\apnumZ` (new Digit from the input stream) is initialized as zero and the `\apNexti` runs the next step of this loop. This step starts from `\apDIVh` (reading one digit from input stream) or directly the `\apDIVi` is repeated. If the remainder from the previous step is calculated as zero (`\apnumE=0`), then we stop prematurely. The `\apDIVj` macro is called at the end of the loop because we need to remove the “rest of the dividend” from the input stream.

apnum.tex

```
402: \def\apDIVi{%
403:   \ifnum\apnumE=0 \apnumC=0 \fi
404:   \ifnum\apnumC>0
405:     \expandafter\apDIVp\XOUT
406:     \advance\apnumC by-4
407:     \apnumZ=0
408:     \expandafter\apNexti
409:   \else
410:     \expandafter\apDIVj
411:   \fi
412: }
413: \def\apDIVj#1!{}%
```

The macro `\apDIVp<interleaved data>@` does the basic setting before the calculation through the expanded `\XOUT` is processed. The `\apDIVxA` includes the “partial dividend” and the `\apDIVxB` includes the “partial divisor”. We need to do `\apDIVxA` over `\apDIVxB` in order to obtain the next digit in the output. This digit is stored in `\apnumA`. The `\apnumX` is the transmission value, the `\apnumB`, `\apnumY` will be the memory of the last two calculated Digits in the remainder. The `\apnumE` will include the maximum of all digits of the new remainder. If it is equal to zero, we can finish the calculation.

The new interleaved data will be stored to the `\apOUT:<num>` macros in similar way as in the `\apMUL` macro. This increases the speed of the calculation. The data `\apnum0`, `\apnumL` and `\apOUT1` for this purpose are initialized.

The `\apDIVp` is started and the tokens `0\apnumZ` are appended to the input stream (i. e. to the expanded `\XOUT`). This zero will be ignored and the `\apnumZ` will be used as a new N_n , i. e. the Digit from the “rest of the dividend”.

apnum.tex

```
414: \def\apDIVp{%
415:   \apnumA=\apDIVxA \divide\apnumA by\apDIVxB
416:   \def\apOUT1{}\apnum0=1 \apnumL=0
417:   \apnumX=0 \apnumB=0 \apnumE=0
418:   \let\apNext=\apDIVq \apNext 0\apnumZ
419: }
```

The macro `\apDIVq<dkkk-1 calculates the Digit of the new remainder N_{k-1} by the formula $N_{k-1} = -A \cdot d_k + R_k - X$ where X is the transmission from the previous Digit. If the result is negative, we need to add minimal number of the form $X \cdot 10000$ in order the result is non-negative. Then the X is new transmission value. The digit N_k is stored in the \apnumB register and then it is added to \apOUT:<num> in the order $d_{k-1} N_{k-1}$. The \apnumY remembers the value of the previous \apnumB. The d_{k-1} is put to the input stream back in order it would be read by the next \apDIVq call.`

`\apDIVh`: 25–27 `\apDIVi`: 25–26 `\apDIVj`: 26 `\apDIVp`: 26 `\apDIVxA`: 23–24, 26–27
`\apDIVxB`: 23–24, 26 `\apDIVq`: 26–27

If $d_{k-1} = 0$ then we are at the end of the remainder calculation and the `\apDIVr` is invoked.

```

420: \def\apDIVq#1#2#3{% B A B
421:   \advance\apnum0 by-1 \ifnum\apnum0=0 \apOUTx \fi
422:   \apnumY=\apnumB
423:   \apnumB=#1\multiply\apnumB by-\apnumA
424:   \advance\apnumB by#2\advance\apnumB by-\apnumX
425:   \ifnum\apnumB<0 \apnumX=\apnumB \advance\apnumX by1
426:   \divide\apnumX by-\apIVbase \advance\apnumX by1
427:   \advance\apnumB by\the\apnumX 0000
428:   \else \apnumX=0 \fi
429:   \expandafter
430:   \edef\csname apOUT:\apOUTn\endcsname{\csname apOUT:\apOUTn\endcsname{#3}{\the\apnumB}}}%
431:   \ifnum\apnumE<\apnumB \apnumE=\apnumB \fi
432:   \ifx@#3\let\apNext=\apDIVr \fi
433:   \apNext{#3}%
434: }
```

The `\apDIVr` macro does the final work after the calculation of new remainder is done. It tests if the remainder is OK, i. e. the transmission from the R_1 calculation is equal to R_0 . If it is true then new Digit `\apnumA` is added to the `\OUT` macro else the `\apnumA` is decreased (the correction) and the calculation of the remainder is run again.

If the calculated Digit and the remainder are OK, then we do following:

- The new `\XOUT` is created from `\apOUT:\langle num \rangle` macros using `\apOUTs` macro.
- The `\apnumA` is saved to the `\OUT`. This is done with care. If the `\apnumD` (where the decimal point is measured from the actual point in the `\OUT`) is in the interval $[0, 4)$ then the decimal point have to be inserted between digits into the next Digit. This is done by `\apDIVt` macro. If the remainder is zero (`\apnumE=0`), then the right trailing zeros are removed from the Digit by the `\apDIVu` and the shift of the `\apnumD` register is calculated from the actual digits. All this calculation is done in `\tmpa` macro. The last step is adding the contents of `\tmpa` to the `\OUT`.
- The `\apnumD` is increased by the number of added digits.
- The new “partial dividend” is created from `\apnumB` and `\apnumY`.

```

435: \def\apDIVr#1#2{%
436:   \ifnum\apnumX=#2 % the calculated Digit is OK, we save it
437:   \edef\XOUT{\expandafter\apOUTs\apOUTl.,}%
438:   \edef\tmpa{\ifnum\apnumF=4 \expandafter\apIVwrite\else \expandafter\the\fi\apnumA}%
439:   \ifnum\apnumD<\apnumF \ifnum\apnumD>-1 \apDIVt \fi\fi %adding dot
440:   \ifx\apNexti\apDIVh \apnumE=1 \fi
441:   \ifnum\apnumE=0 \apDIVu % removing zeros
442:   \advance\apnumD by-\apNUMdigits\tmpa \relax
443:   \else \advance\apnumD by-\apnumF \apnumF=4 \fi
444:   \edef\OUT{\OUT\tmpa}% save the Digit
445:   \edef\apDIVxA{\the\apnumB\apIVwrite\apnumY}% next partial dividend
446:   \else % we need do correction and run the remainder calculation again
447:   \advance\apnumA by-1 \apnumX=0 \apnumB=0 \apnumE=0
448:   \def\apOUTl{}\apnum0=1 \apnumL=0
449:   \def\apNext{\let\apNext=\apDIVq
450:   \expandafter\apNext\expandafter0\expandafter\apnumZ\XOUT}%
451:   \expandafter\apNext
452:   \fi
453: }
```

The `\apDIVt` macro inserts the dot into digits quartet (less than four digits are allowed too) by the `\apnumD` value. This value is assumed in the interval $[0, 4)$. The expandable macro `\apIVdot\langle shift \rangle\langle data \rangle` is used for this purpose. The result from this macro has to be expanded twice.

```

454: \def\apDIVt{\edef\tmpa{\apIVdot\apnumD\tmpa}\edef\tmpa{\tmpa}}
```

The `\apDIVu` macro removes trailing zeros from the right and removes the dot, if it is the last token of the `\tmpa` after removing zeros. It uses expandable macros `\apREMzeros\langle data \rangle` and `\apREMDotr\langle data \rangle`.

```
455: \def\apDIVu{\edef\tmpa{\apREMzerosR\tmpa}\edef\tmpa{\apREMdotR\tmpa}}
```

apnum.tex

The rest of the code concerned with the division does an extraction of the last remainder from the data and this value is saved to the `\XOUT` macro in human readable form. The `\apDIVv` macro is called repeatedly on the special format of the `\XOUT` macro and the new `\XOUT` is created. The trailing zeros from right are ignored by the `\apDIVw`.

```
456: \def\apDIVv#1#2{\apnumX=#2
457:   \ifx@#1\apDIVw{.\apIVwrite\apnumX}\else\apDIVw{\apIVwrite\apnumX}\expandafter\apDIVv\fi
458: }
459: \def\apDIVw#1{%
460:   \ifx\XOUT\empty \ifnum\apnumX=0
461:     \else \edef\tmpa{#1}\edef\XOUT{\apREMzerosR\tmpa\XOUT}%
462:     \fi
463:   \else \edef\XOUT{#1\XOUT}\fi
464: }
```

apnum.tex

2.7 Power to the Integer

The `\apPOW` macro does the power to the integer exponent only. The `\apPOWx` is equivalent to `\apPOW` and it is used in `\evaldef` macro for the \wedge operator. If you want to redefine the meaning of the \wedge operator then redefine the `\apPOWx` sequence.

```
468: \def\apPOW{\relax \apPPab\apPOWa} \let\apPOWx=\apPOW % for usage as ^ operator
```

apnum.tex

We can implement the power to the integer as repeated multiplications. This is simple but slow. The goal of this section is to present the power to the integer with some optimizations.

Let a is the base of the powering computation and $d_1, d_2, d_3, \dots, d_n$ are binary digits of the exponent (in reverse order). Then

$$p = a^{1d_1 + 2d_2 + 2^2d_3 + \dots + 2^{n-1}d_n} = (a^1)^{d_1} \cdot (a^2)^{d_2} \cdot (a^{2^2})^{d_3} \cdot (a^{2^{n-1}})^{d_n}.$$

If $d_i = 0$ then z^{d_i} is one and this can be omitted from the queue of multiplications. If $d_i = 1$ then we keep z^{d_i} as z in the queue. We can see from this that the p can be computed by the following algorithm:

```
(* "a" is initialized as the base, "e" as the exponent *)
p := 1;
while (e>0) {
  if (e%2) p := p*a;
  e := e/2;
  if (e>0) a := a*a;
}
(* "p" includes the result *)
```

The macro `\apPOWa` does the following work.

- After using `\apPPab` the base parameter is saved in `\tmpa` and the exponent is saved in `\tmpb`.
- In trivial cases, the result is set without any computing (lines 470 and 471).
- If the exponent is non-integer or it is too big then the error message is printed and the rest of the macro is skipped by the `\apPOWe` macro (lines 473 to 476).
- The `\apE` is calculated from `\apEa` (line 477).
- The sign of the result is negative only if the `\tmpb` is odd and base is negative (line 479).
- The number of digits after decimal point for the result is calculated and saved to `\apnumD`. The total number of digits of the base is saved to `\apnumC`. (line 480).
- The first Digit of the base needn't to include all four digits, but other Digits do it. The similar trick as in `\apMULa` is used here (lines 482 to 483).
- The base is saved in interleaved reversed format (like in `\apMULa`) into the `\OUT` macro by the `\apMULb` macro. Let it be the a value from our algorithm described above (lines 484 and 485).

`\XOUT`: 6, 5, 14, 23–28, 32, 37–38, 45 `\apDIVv`: 23–24, 28 `\apDIVw`: 28 `\apPOW`: 6, 12, 14, 28, 36,
40, 42, 45, 47, 50 `\apPOWx`: 11–12, 28, 47–48 `\apPOWa`: 28–30

- The initial value of $p = 1$ from our algorithm is set in interleaved format into `\tmpc` macro (line 486).
- The main loop described above is processed by `\apPOWb` macro. (line 487).
- The result in `\tmpc` is converted into human readable form by the `\apPOWg` macro and it is stored into the `\OUT` macro (line 488).
- If the result is negative or decimal point is needed to print then use simple conversion of the `\OUT` macro (adding minus sign) or using `\apROLLa` macro (lines 489 and 490).
- If the exponent is negative then do the $1/r$ calculation, where r is previous result (line 491).

```

469: \def\apPOWa{%
470:   \ifnum\apSIGNa=0 \def\OUT{0}\apSIGN=0 \apE=0 \else
471:     \ifnum\apSIGNb=0 \def\OUT{1}\apSIGN=1 \apE=0 \else
472:       \apDIG\tmpb\apnumB
473:       \ifnum\apnumB>0 \apERR{POW: non-integer exponent is not implemented yet}\apPOWe\fi
474:       \ifnum\apEb=0 \else \apERR{POW: the E notation of exponent isn't allowed}\apPOWe\fi
475:       \ifnum\apnumD>8 \apERR{POW: too big exponent.
476:         Do you really need about 10the\apnumD\space digits in output?}\apPOWe\fi
477:       \apE=\apEa \multiply\apE by\tmpb\relax
478:       \apSIGN=\apSIGNa
479:       \ifodd\tmpb \else \apSIGN=1 \fi
480:       \apDIG\tmpa\apnumA \apnumC=\apnumA \advance\apnumC by\apnumD
481:       \apnumD=\apnumA \multiply\apnumD by\tmpb
482:       \apIVmod \apnumC \apnumA
483:       \edef\tmpc{\ifcase\apnumA\or{}\or{}\or{}\or{}\or{}\fi}\def\OUT{}%
484:       \expandafter\expandafter\expandafter \apMULb \expandafter \tmpc \tmpa @@@@%
485:       \edef\OUT{*\OUT}% \OUT := \tmpa in interleaved format
486:       \def\tmpc{*.1*}%
487:       \apnumE=\tmpb\relax \apPOWb
488:       \expandafter\apPOWg \tmpc % \OUT := \tmpc in human raedable form
489:       \ifnum\apnumD=0 \ifnum \apSIGN<0 \edef\OUT{-\OUT}\fi
490:       \else \def\tmpc{-\apnumD}\apnumG=\apSIGN \expandafter\apROLLa\OUT.@\OUT\fi
491:       \ifnum\apSIGNb<0 \apPPab\apDIVa 1\OUT \fi
492:       \relax
493:     \fi\fi
494: }

```

The macro `\apPOwb` is the body of the loop in the algorithm described above. The code part after `\ifodd\apnumE` does $p := p \cdot a$. In order to do this, we need to convert `\OUT` (where a is stored) into normal format using `\apPOwd`. The result is saved in `\tmpb`. Then the multiplication is done by `\apMULd` and the result is normalized by the `\apPOwn` macro. Because `\apMULd` works with `\OUT` macro, we temporary set `\tmpc` to `\OUT`.

The code part after `\ifnum\apnumE<0` does `a := a*a` using the `\apPOWt` macro. The result is normalized by the `\apPOWn` macro.

```

495: \def\apPOwb{%
496:   \ifodd\apnumE   \def\tmpb{}\expandafter\apPOwd\OUT
497:                   \let\tmpd=\OUT \let\OUT=\tmpc
498:                   \expandafter\apMULd \tmpb@\expandafter\apPOwn\OUT@%
499:                   \let\tmpc=\OUT \let\OUT=\tmpd
500:   \fi
501:   \divide\apnumE by2
502:   \ifnum\apnumE>0 \expandafter\apPOwt\OUT \expandafter\apPOwn\OUT@%
503:                   \expandafter\apPOwb
504:   \fi
505: }

```

The macro `\apPOwd` *`\apPOwd`* *`⟨initialized interleaved reversed format⟩`* extracts the Digits from its argument and saves them to the `\tmpb` macro.

```

506: \def\apPOwd#1#2{% \apPOwd <spec format> => \tmpb (in simple reverse format)
507:   \ifx#1\expandafter\apPOwd \else
508:     \edef\tmpb{\tmpb{#1}}%
509:     \ifx#2\else \expandafter\expandafter\apPOwd\fi

```

\apPOWb: 29 \apPOWd: 29

```
510: \fi
511: }
```

The `\apPOWe` macro skips the rest of the body of the `\apPOWa` macro to the `\relax`. It is used when `\errmessage` is printed.

```
512: \def\apPOWe#1\relax{\fi}
```

apnum.tex

The `\apPOWg` macro provides the conversion from interleaved reversed format to the human readable form and save the result to the `\OUT` macro. It ignores the first two elements from the format and runs `\apPOWh`.

```
513: \def\apPOWg#1#2{\def\OUT{}\apPOWh} % conversion to the human readable form
514: \def\apPOWh#1#2{\apnumA=#1
515:   \ifx##2\edef\OUT{\the\apnumA\OUT}\else \edef\OUT{\apIVwrite\apnumA\OUT}\expandafter\apPOWh\fi
516: }
```

apnum.tex

The normalization to the initialized interleaved format of the `\OUT` is done by the `\apPOWn` $\langle data \rangle$ macro. The `\apPOWna` reads the first part of the $\langle data \rangle$ (to the first `*`, where the Digits are non-interleaved). The `\apPOWnn` reads the second part of $\langle data \rangle$ where the Digits of the result are interleaved with the digits of the old coefficients. We need to set the result as a new coefficients and prepare zeros between them for the new calculation. The dot after the first `*` is not printed (the zero is printed instead it) but it does not matter because this token is simply ignored during the calculation.

```
517: \def\apPOWn#1{\def\OUT{*}\apPOWna}
518: \def\apPOWna#1{\ifx##1\expandafter\apPOWnn\else \edef\OUT{\OUT0{#1}}\expandafter\apPOWna\fi}
519: \def\apPOWnn#1#2{\ifx##1\edef\OUT{\OUT*}\else\edef\OUT{\OUT0{#1}}\expandafter\apPOWnn\fi}
```

apnum.tex

The powering to two (`\OUT:=\OUT^2`) is provided by the `\apPOWt` $\langle data \rangle$ macro. The macro `\apPOWu` is called repeatedly for each `\apnumA=Digit` from the $\langle data \rangle$. One line of the multiplication scheme is processed by the `\apPOWv` $\langle data \rangle$ macro. We can call the `\apMULe` macro here but we don't do it because a slight optimization is used here. You can try to multiply the number with digits `abcd` by itself in the mirrored multiplication scheme. You'll see that first line includes `a^2 2ab 2ac 2ad`, second line is intended by two columns and includes `b^2 2bc 2bd`, next line is intended by next two columns and includes `c^2 2cd` and the last line is intended by next two columns and includes only `d^2`. Such calculation is slightly shorter than normal multiplication and it is implemented in the `\apPOWv` macro.

```
520: \def\apPOWt#1#2{\apPOWu} % power to two
521: \def\apPOWu#1#2{\apnumA=#1
522:   \expandafter\apPOWv\OUT
523:   \ifx##2\else \expandafter\apPOWu\fi
524: }
525: \def\apPOWv#1#2#3#4{\def\apOUT1{}\apnum0=1 \apnumL=0
526:   \apnumB=\apnumA \multiply\apnumB by\apnumB \multiply\apnumA by2
527:   \ifx##4\else\advance\apnumB by#4 \fi
528:   \ifx\apnumB<\apIVbase \apnumX=0 \else \apIVtrans \fi
529:   \edef\OUT{#1{#2}{\the\apnumB}*}%
530:   \ifx##4\apMULf0*\else\expandafter\apMULf\fi
531: }
```

apnum.tex

2.8 apROLL, apROUND and apNORM Macros

The macros `\apROLL`, `\apROUND` and `\apNORM` are implemented by `\apROLLa`, `\apROUNDa` and `\apNORMa` macros with common format of the parameter text: $\langle expanded\ sequence \rangle . @ \langle sequence \rangle$ where $\langle expanded\ sequence \rangle$ is the expansion of the macro $\langle sequence \rangle$ (given as first parameter of `\apROLL`, `\apROUND` and `\apNORM`, but without optionally minus sign. If there was the minus sign then `\apnumG=-1` else `\apnumG=1`. This preparation of the parameter $\langle sequence \rangle$ is done by the `\apPPs` macro. The second parameter of the macros `\apROLL`, `\apROUND` and `\apNORM` is saved to the `\tmpc` macro.

<code>\apPOWe</code> : 28–30	<code>\apPOWg</code> : 29–30	<code>\apPOWh</code> : 30	<code>\apPOWn</code> : 29–30	<code>\apPOWna</code> : 30	<code>\apPOWnn</code> : 30
<code>\apPOWt</code> : 29–30	<code>\apPOWu</code> : 30	<code>\apPOWv</code> : 30	<code>\apROLL</code> : 5, 14, 30–31, 33, 39, 41, 45, 50		
<code>\apROUND</code> : 5, 14, 30, 32, 37–38, 40–43, 45–47, 50	<code>\apNORM</code> : 5, 14, 30, 33, 50				

`\apROLLa` $\langle param \rangle . @ \langle sequence \rangle$ shifts the decimal point of the $\langle param \rangle$ by `\tmpc` positions to the right (or to the left, if `\tmpc` is negative) and saves the result to the $\langle sequence \rangle$ macro. The `\tmpc` value is saved to the `\apnumA` register and the `\apROLLc` is executed if we need to shift the decimal point to left. Else `\apROLLg` is executed.

apnum.tex

```
535: \def\apROLL{\apPPs\apROLLa}
536: \def\apROLLa{\apnumA=\tmpc\relax \ifnum\apnumA<0 \expandafter\apROLLc\else \expandafter\apROLLg\fi}
```

The `\apROLLc` $\langle param \rangle . @ \langle sequence \rangle$ shifts the decimal point to left by the `-\apnumA` decimal digits. It reads the tokens from the input stream until the dot is found using `\apROLLd` macro. The number of such tokens is set to the `\apnumB` register and tokens are saved to the `\tmpc` macro. If the dot is found then `\apROLLe` does the following: if the number of read tokens is greater then the absolute value of the $\langle shift \rangle$, then the number of positions from the most left digit of the number to the desired place of the dot is set to the `\apnumA` register a the dot is saved to this place by `\apROLLi` $\langle parameter \rangle . @ \langle sequence \rangle$. Else the new number looks like .000123 and the right number of zeros are saved to the $\langle sequence \rangle$ using the `\apADDzeros` macro and the rest of the input stream (including expanded `\tmpc` returned back) is appended to the macro $\langle sequence \rangle$ by the `\apROLLf` $\langle param \rangle . @$ macro.

apnum.tex

```
537: \def\apROLLc{\edef\tmpc{\}\edef\tmpd{\ifnum\apnumG<0-\fi}\apnumB=0 \apROLLd}
538: \def\apROLLd#1{%
539:   \ifx.#1\expandafter\apROLLe
540:   \else \edef\tmpc{\tmpc#1}%
541:     \advance\apnumB by1
542:     \expandafter\apROLLd
543:   \fi
544: }
545: \def\apROLLe#1{\ifx@#1\edef\tmpc{\tmpc.@}\else\edef\tmpc{\tmpc#1}\fi
546:   \advance\apnumB by\apnumA
547:   \ifnum\apnumB<0
548:     \apnumZ=-\apnumB \edef\tmpd{\tmpd.}\apADDzeros\tmpd
549:     \expandafter\expandafter\expandafter\apROLLf\expandafter\tmpc
550:   \else
551:     \apnumA=\apnumB
552:     \expandafter\expandafter\expandafter\apROLLi\expandafter\tmpc
553:   \fi
554: }
555: \def\apROLLf#1.@#2{\edef#2{\tmpd#1}}
```

The `\apROLLg` $\langle param \rangle . @ \langle sequence \rangle$ shifts the decimal point to the right by `\apnumA` digits starting from actual position of the input stream. It reads tokens from the input stream by the `\apROLLh` and saves them to the `\tmpd` macro where the result will be built. When dot is found the `\apROLLi` is processed. It reads next tokens and decreases the `\apnumA` by one for each token. It ends (using `\apROLLj``\apROLLk`) when `\apnumA` is equal to zero. If the end of the input stream is reached (the `@` character) then the zero is inserted before this character (using `\apROLLj``\apROLLi0@`). This solves the situations like 123, $\langle shift \rangle = 2$, \rightarrow 12300.

apnum.tex

```
556: \def\apROLLg#1{\edef\tmpd{\ifnum\apnumG<0-\fi}\ifx.#1\apnumB=0 \else\apnumB=1 \fi \apROLLh#1}
557: \def\apROLLh#1{\ifx.#1\expandafter\apROLLi\else \edef\tmpd{\tmpd#1}\expandafter\apROLLh\fi}
558: \def\apROLLi#1{\ifx.#1\expandafter\apROLLi\else
559:   \ifnum\apnumA>0 \else \apROLLj \apROLLk#1\fi
560:   \ifx@#1\apROLLj \apROLLi0@\fi
561:   \advance\apnumA by-1
562:   \ifx0#1\else \apnumB=1 \fi
563:   \ifnum\apnumB>0 \edef\tmpd{\tmpd#1}\fi
564:   \expandafter\apROLLi\fi
565: }
```

The `\apROLLg` macro initializes `\apnumB=1` if the $\langle param \rangle$ doesn't begin by dot. This is a flag that all digits read by `\apROLLi` have to be saved. If the dot begins, then the number can look like .000123 (before moving the dot to the right) and we need to ignore the trailing zeros. The `\apnumB` is equal to zero in such case and this is set to 1 if here is first non-zero digit.

`\apROLLa`: 18, 23–24, 29–33 `\apROLLc`: 31 `\apROLLd`: 31 `\apROLLe`: 31 `\apROLLf`: 31
`\apROLLg`: 31 `\apROLLh`: 31 `\apROLLi`: 31–32

The `\apROLLj` macro closes the conditionals and runs its parameter separated by `\fi`. It skips the rest of the `\apROLLi` macro too.

apnum.tex

```
566: \def\apROLLj#1\fi#2\apROLLi\fi{\fi\fi#1}
```

The macro `\apROLLk` puts the decimal point to the `\tmpd` at current position (using `\apROLLn`) if the input stream is not fully read. Else it ends the processing. The result is an integer without decimal digit in such case.

apnum.tex

```
567: \def\apROLLk#1{\ifx@#1\expandafter\apROLLo\expandafter@else
568:   \def\tmpc{}\apnumB=0 \expandafter\apROLLn\expandafter#1\fi
569: }
```

The macro `\apROLLn` reads the input stream until the dot is found. Because we read now the digits after a new position of the decimal point we need to check situations of the type 123.000 which is needed to be written as 123 without decimal point. This is a reason of a little complication. We save all digits to the `\tmpc` macro and calculate the sum of such digits in `\apnumB` register. If this sum is equal to zero then we don't append the `\tmpc` to the `\tmpd`. The macro `\apROLLn` is finished by the `\apROLLo` `@sequence` macro, which removes the last token from the input stream and defines `sequence` as `\tmpd`.

apnum.tex

```
570: \def\apROLLn#1{%
571:   \ifx.#1\ifnum\apnumB>0 \edef\tmpd{\tmpd.\tmpc}\fi \expandafter\apROLLo
572:   \else \edef\tmpc{\tmpc#1}\advance\apnumB by#1 \expandafter\apROLLn
573:   \fi
574: }
575: \def\apROLLo@#1{\let#1=\tmpd}
```

The macro `\apROUNDa` `<param>.@<sequence>` rounds the number given in the `<param>`. The number of digits after decimal point `\tmpc` is saved to `\apnumD`. If this number is negative then `\apROUNDe` is processed else the `\apROUNDb` reads the `<param>` to the decimal point and saves this part to the `\tmpc` macro. The `\tmpd` macro (where the rest after decimal point of the number will be stored) is initialized to empty and the `\apROUNDc` is started. This macro reads one token from input stream repeatedly until the number of read tokens is equal to `\apnumD` or the stop mark `@` is reached. All tokens are saved to `\tmpd`. Then the `\apROUNDd` macro reads the rest of the `<param>`, saves it to the `\XOUT` macro and defines `sequence` (i. e. #2) as the rounded number.

apnum.tex

```
577: \def\apROUND{\apPPs\apROUNDa}
578: \def\apROUNDa{\apnumD=\tmpc\relax
579:   \ifnum\apnumD<0 \expandafter\apROUNDe
580:   \else \expandafter\apROUNDb
581:   \fi
582: }
583: \def\apROUNDb#1.{\edef\tmpc{#1}\apnumX=0 \def\tmpd{}\let\apNext=\apROUNDc \apNext}
584: \def\apROUNDc#1{\ifx@#1\def\apNext{\apROUNDd.@}%
585:   \else \advance\apnumD by-1
586:   \ifnum\apnumD<0 \def\apNext{\apROUNDd#1}%
587:   \else \ifx.#1\else \advance\apnumX by#1 \edef\tmpd{\tmpd#1}\fi
588:   \fi
589:   \fi \apNext
590: }
591: \def\apROUNDd#1.@#2{\def\XOUT{#1}\edef\XOUT{\apREMzerosR\XOUT}%
592:   \ifnum\apnumX=0 \def\tmpd{}\fi
593:   \ifx\tmpd\empty
594:     \ifx\tmpc\empty \def#2{0}%
595:     \else \edef#2{\ifnum\apnumG<0-\fi\tmpc}\fi
596:   \else\edef#2{\ifnum\apnumG<0-\fi\tmpc.\tmpd}\fi
597: }
```

The macro `\apROUNDe` solves the “less standard” problem when rounding to the negative digits after decimal point `\apnumD`, i. e. we need to set `-\apnumD` digits before decimal point to zero. The solution is to remove the rest of the input stream, use `\apROLLa` to shift the decimal point left by

`\apROLLj`: 31–32 `\apROLLk`: 31–32 `\apROLLn`: 32 `\apROLLo`: 32 `\apROUNDa`: 14, 30, 32–33
`\apROUNDb`: 32 `\apROUNDc`: 32 `\apROUNDd`: 32 `\apROUNDe`: 32–33

–\apnumD positions, use \apROUNDa to remove all digits after decimal point and shift the decimal point back to its previous place.

```

598: \def\apROUND#1.#2{\apnumC=\apnumD
599:   \apPPs\apROLLa#2{\apnumC}\apPPs\apROUNDa#2{0}\apPPs\apROLLa#2{-\apnumC}%
600: }

```

apnum.tex

The macro \apNORMa redefines the $\langle sequence \rangle$ in order to remove minus sign because the \apDIG macro uses its parameter without this sign. Then the \apNORMb $\langle sequence \rangle \langle parameter \rangle @$ is executed where the dot in the front of the parameter is tested. If the dot is here then the \apDIG macro measures the digits after decimal point too and the \apNORMc is executed (where the \apROLLa shifts the decimal point from the right edge of the number). Else the \apDIG macro doesn't measure the digits after decimal point and the \apNORMd is executed (where the \apROLLa shifts the decimal point from the left edge of the number).

```

601: \def\apNORM{\apPPs\apNORMa}
602: \def\apNORMa#1.#2{\ifnum\apnumG<0 \def#2{#1}\fi \expandafter\apNORMb\expandafter#2\tmpc@}
603: \def\apNORMb#1#2#3@{%
604:   \ifx.#2\apnumC=#3\relax \apDIG#1\apnumA \apNORMc#1%
605:   \else \apnumC=#2#3\relax \apDIG#1\relax \apNORMd#1%
606:   \fi
607: }
608: \def\apNORMc#1{\advance\apE by-\apnumA \advance\apE by\apnumC
609:   \def\tmpc{-\apnumC}\expandafter\apROLLa#1.#1%
610: }
611: \def\apNORMd#1{\advance\apE by\apnumD \advance\apE by-\apnumC
612:   \def\tmpc{\apnumC}\expandafter\apROLLa\expandafter.#1.#1%
613: }

```

apnum.tex

The macro \apEadd $\langle sequence \rangle$ adds E in scientific format into $\langle sequence \rangle$ macro and \apEnum $\langle sequence \rangle$ normalizes the number in the $\langle sequence \rangle$. After processing these macros the \apE register is set to zero.

```

614: \def\apEadd#1{\ifnum\apE=0 \else\edef#1{#1E\ifnum\apE>0+\fi\the\apE}\apE=0 \fi}
615: \def\apEnum#1{\ifnum\apE=0 \else\apROLL#1\apE \apE=0 \fi}

```

apnum.tex

2.9 Miscellaneous Macros

The macro \apEND closes the \begingroup group, but keeps the values of \OUT macro and \apSIGN, \apE registers.

```

619: \def\apEND{\global\let\apENDx=\OUT
620:   \edef\tmpb{\apSIGN=\the\apSIGN \apE=\the\apE}%
621:   \expandafter\endgroup \tmpb \let\OUT=\apENDx
622: }

```

apnum.tex

The macro \apDIG $\langle sequence \rangle \langle register or relax \rangle$ reads the content of the macro $\langle sequence \rangle$ and counts the number of digits in this macro before decimal point and saves it to \apnumD register. If the macro $\langle sequence \rangle$ includes decimal point then it is redefined with the same content but without decimal point. The numbers in the form .00123 are replaced by 123 without zeros, but \apnumD=-2 in this example. If the second parameter of the \apDIG macro is \relax then the number of digits after decimal point isn't counted. Else the number of these digits is stored to the given $\langle register \rangle$.

The macro \apDIG is developed in order to do minimal operations over a potentially long parameters. It assumes that $\langle sequence \rangle$ includes a number without $\langle sign \rangle$ and without left trailing zeros. This is true after parameter preparation by the \apPPab macro.

The macro \apDIG prepares an incrementation in \tmpc if the second parameter $\langle register \rangle$ isn't \relax. It initializes \apnumD and $\langle register \rangle$. It runs \apDIGa $\langle data \rangle . @ \langle sequence \rangle$ which increments the \apnumD until the dot is found. Then the \apDIGb is executed (if there are no digits before dot) or the \apDIGc is called (if there is at least one digit before dot). The \apDIGb ignores zeros immediately

```

\apNORMa: 30, 33   \apNORMb: 33   \apNORMc: 33   \apNORMd: 33   \apEadd: 4, 5, 33, 37
\apEnum: 4, 5, 33, 37–38, 40, 45–47   \apEND: 10, 12, 33, 36–42, 44, 46–47   \apDIG: 15–16, 18–19,
23–24, 29, 33–34, 39, 41, 46   \apDIGa: 34   \apDIGb: 33–34

```

after dot. The `\apDIGc` reads the rest of the $\langle data \rangle$ to the #1 and saves it to the `\tmpd` macro. It runs the counter over this $\langle data \rangle$ `\apDIGd` $\langle data \rangle$ @ only if it is desired (`\tmpc` is non-empty). Else the `\apDIGe` is executed. The `\apDIGe` $\langle dot \text{ or nothing } \rangle$ @ $\langle sequence \rangle$ redefines $\langle sequence \rangle$ if it is needed. Note, that #1 is empty if and only if the $\langle data \rangle$ include no dot (first dot was reached as the first dot from `\apDIG`, the second dot from `\apDIG` was a separator of #1 in `\apDIGc` and there is nothing between the second dot and the @ mark. The $\langle sequence \rangle$ isn't redefined if it doesn't include a dot. Else the sequence is set to the `\tmpd` (the rest after dot) if there are no digits before dot. Else the sequence is redefined using expandable macro `\apDIGf`.

apnum.tex

```

623: \def\apDIG#1#2{\ifx\relax#2\def\tmpc{}\else #2=0 \def\tmpc{\advance#2 by1 }\fi
624:   \apnumD=0 \expandafter\apDIGa#1..@#1%
625: }
626: \def\apDIGa#1{\ifx.#1\csname apDIG\ifnum\apnumD>0 c\else b\fi\expandafter\endcsname
627:   \else \advance\apnumD by1 \expandafter\apDIGa\fi}
628: \def\apDIGb#1{%
629:   \ifx0#1\advance\apnumD by-1 \tmpc \expandafter\apDIGb
630:   \else \expandafter\apDIGc \expandafter#1\fi
631: }
632: \def\apDIGc#1.{\def\tmpd{#1}%
633:   \ifx\tmpc\empty \let\apNext=\apDIGe
634:   \else \def\apNext{\expandafter\apDIGd\tmpd@}%
635:   \fi \apNext
636: }
637: \def\apDIGd#1{\ifx@#1\expandafter\apDIGe \else \tmpc \expandafter\apDIGd \fi}
638: \def\apDIGe#1@#2{%
639:   \ifx@#1@ \else % #1=empty <=> the param has no dot, we need to do nothing
640:     \ifnum\apnumD>0 \edef#2{\expandafter\apDIGf#2@}% the dot plus digits before dot
641:     \else \let#2=\tmpd % there are only digits after dot, use \tmpd
642:     \fi\fi
643: }
644: \def\apDIGf#1.#2@{#1#2}

```

The macro `\apIVread` $\langle sequence \rangle$ reads four digits from the macro $\langle sequence \rangle$, sets `\apnumX` as the Digit consisting from read digits and removes the read digits from $\langle sequence \rangle$. It internally expands $\langle sequence \rangle$, adds the `\apNL` marks and runs `\apIVreadA` macro which sets the `\apnumX` and redefines $\langle sequence \rangle$.

The usage of the `\apNL` as a stop-marks has the advantage: they act as simply zero digits in the comparison but we can ask by `\ifx` if this stop mark is reached. The #5 parameter of `\apIVreadA` is separated by first occurrence of `\apNL`, i. e. the rest of the macro $\langle sequence \rangle$ is here.

apnum.tex

```

646: \def\apNL{0}
647: \def\apIVread#1{\expandafter\apIVreadA#1\apNL\apNL\apNL\apNL\apNL@#1}
648: \def\apIVreadA#1#2#3#4#5\apNL#6@#7{\apnumX=#1#2#3#4\relax \def#7{#5}}

```

The macro `\apIVreadX` $\langle num \rangle$ $\langle sequence \rangle$ acts similar as `\apIVread` $\langle sequence \rangle$, but only $\langle num \rangle$ digits are read. The $\langle num \rangle$ is expected in the range 0 to 4. The macro prepares the appropriate number of empty parameters in `\tmpc` and runs `\apIVreadA` with these empty parameters inserted before the real body of the $\langle sequence \rangle$.

apnum.tex

```

649: \def\apIVreadX#1#2{\edef\tmpc{\ifcase#1\{}{}0\or\{}{}1\or\{}{}2\or\{}{}3\or\{}{}4\or\{}{}5\or\{}{}6\or\{}{}7\or\{}{}8\or\{}{}9\fi}%
650:   \expandafter\expandafter\expandafter\apIVreadA\expandafter\tmpc#2\apNL\apNL\apNL\apNL\apNL@#2%
651: }

```

The macro `\apIVwrite` $\langle num \rangle$ expands the digits from $\langle num \rangle$ register. The number of digits are four. If the $\langle num \rangle$ is less than 1000 then left zeros are added.

apnum.tex

```

652: \def\apIVwrite#1{\ifnum#1<1000 0\ifnum#1<100 0\ifnum#1<10 0\fi\fi\fi\the#1}

```

The macro `\apIVtrans` calculates the transmission for the next Digit. The value (greater or equal 10000) is assumed to be in `\apnumB`. The new value less than 10000 is stored to `\apnumB` and the

<code>\apDIGc</code> : 33–34	<code>\apDIGd</code> : 34	<code>\apDIGe</code> : 34	<code>\apDIGf</code> : 34	<code>\apIVread</code> : 16–17, 23–25, 34
<code>\apIVreadA</code> : 34	<code>\apNL</code> : 16–17, 24–26, 34	<code>\apIVreadX</code> : 23–24, 34	<code>\apIVwrite</code> : 18, 21, 23–24,	
27–28, 30, 34	<code>\apIVtrans</code> : 20–21, 30, 35			

transmission value is stored in `\apnumX`. The constant `\apIVbase` is used instead of literal 10000 because it is quicker.

```
apnum.tex
654: \mathchardef\apIVbase=10000
655: \def\apIVtrans{\apnumX=\apnumB \divide\apnumB by\apIVbase \multiply\apnumB by-\apIVbase
656:   \advance\apnumB by\apnumX \divide\apnumX by\apIVbase
657: }
```

The macro `\apIVmod` $\langle length \rangle \langle register \rangle$ sets $\langle register \rangle$ to the number of digits to be read to the first Digit, if the number has $\langle length \rangle$ digits in total. We need to read all Digits with four digits, only first Digit can be shorter.

```
apnum.tex
658: \def\apIVmod#1#2#3#4#5{\divide#2by4 \multiply#2by-4 \advance#2by#1\relax}
```

The macro `\apIVdot` $\langle num \rangle \langle param \rangle$ adds the dot into $\langle param \rangle$. Let $K = \langle num \rangle$ and F is the number of digits in the $\langle param \rangle$. The macro expects that $K \in [0, 4)$ and $F \in (0, 4]$. The macro inserts the dot after K -th digit if $K < F$. Else no dot is inserted. It is expandable macro, but two full expansions are needed. After first expansion the result looks like `\apIVdotA` $\langle dots \rangle \langle param \rangle \dots @$ where $\langle dots \rangle$ are the appropriate number of dots. Then the `\apIVdotA` reads the four tokens (maybe the generated dots), ignores the dots while printing and appends the dot after these four tokens, if the rest #5 is non-empty.

```
apnum.tex
662: \def\apIVdot#1#2{\noexpand\apIVdotA\ifcase#1...\or...\or...\or...\fi #2...\@}
663: \def\apIVdotA#1#2#3#4#5.#6@{\ifx.#1\else#1\fi
664:   \ifx.#2\else#2\fi \ifx.#3\else#3\fi \ifx.#4\else#4\fi \ifx.#5\else.#5\fi
665: }
```

The expandable macro `\apNUMdigits` $\{ \langle param \rangle \}$ expands (using the `\apNUMdigitsA` macro) to the number of digits in the $\langle param \rangle$. We assume that maximal number of digits will be four.

```
apnum.tex
666: \def\apNUMdigits#1{\expandafter\apNUMdigitsA#1@@@!}
667: \def\apNUMdigitsA#1#2#3#4#5!{\ifx@#4\ifx@#3\ifx@#2\ifx@#1\else1\fi \else2\fi \else3\fi \else4\fi}
```

The macro `\apADDzeros` $\langle sequence \rangle$ adds `\apnumZ` zeros to the macro $\langle sequence \rangle$.

```
apnum.tex
669: \def\apADDzeros#1{\edef#1{#10}\advance\apnumZ by-1
670:   \ifnum\apnumZ>0 \expandafter\apADDzeros\expandafter#1\fi
671: }
```

The expandable macro `\apREMzerosR` $\{ \langle param \rangle \}$ removes right trailing zeros from the $\langle param \rangle$. It expands to `\apREMzerosRa` $\langle param \rangle @@@!$. The macro `\apREMzerosRa` reads all text terminated by `@` to #1. This termination zero can be the most right zero of the $\langle param \rangle$ (then #2 is non-empty) or $\langle param \rangle$ hasn't such zero digit (then #2 is empty). If #2 is non-empty then the `\apREMzerosRa` is expanded again in the recursion. Else `\apREMzerosRb` removes the stop-mark `@` and the expansion is finished.

```
apnum.tex
672: \def\apREMzerosR#1{\expandafter\apREMzerosRa#1@@@!}
673: \def\apREMzerosRa#10@#2!{\ifx!#2!\apREMzerosRb#1\else\apREMzerosRa#10@@!\fi}
674: \def\apREMzerosRb#1@{#1}
```

The expandable macro `\apREMDotR` $\{ \langle param \rangle \}$ removes right trailing dot from the $\langle param \rangle$ if exists. It expands to `\apREMDotRa` and works similarly as the `\apREMzerosR` macro.

```
apnum.tex
675: \def\apREMDotR#1{\expandafter\apREMDotRa#1@.@!}
676: \def\apREMDotRa#1.@#2!{\ifx!#2!\apREMzerosRb#1\else#1\fi}
```

The `\apREMfirst` $\langle sequence \rangle$ macro removes the first token from the $\langle sequence \rangle$ macro. It can be used for removing the “minus” sign from the “number-like” macros.

```
apnum.tex
678: \def\apREMfirst#1{\expandafter\apREMfirsta#1@#1}
679: \def\apREMfirsta#1#2@#3{\def#3{#2}}
```

`\apIVbase`: 17, 20–21, 27, 30, 35 `\apIVmod`: 15–16, 19, 24, 29, 35 `\apIVdot`: 21, 27, 35
`\apIVdotA`: 35 `\apNUMdigits`: 21, 27, 35 `\apNUMdigitsA`: 35 `\apADDzeros`: 16, 19, 23–24, 31, 35
`\apREMzerosR`: 18, 27–28, 32, 35 `\apREMzerosRa`: 35 `\apREMzerosRb`: 35 `\apREMDotR`: 27–28, 35
`\apREMDotRa`: 35 `\apREMfirst`: 6, 35, 37, 40, 45–46

The writing to the `\OUT` in the `\apMUL`, `\apDIV` and `\apPOW` macros is optimized, which decreases the computation time with very large numbers ten times and more. We can do simply `\edef\OUT{\OUT<something>}` instead of

```
\expandafter\edef\csname apOUT:\apOUTn\endcsname
{\csname apOUT:\apOUTn\endcsname<something>}%
```

but `\edef\OUT{\OUT<something>}` is typically processed very often over possibly very long macro (many thousands of tokens). It is better to do `\edef` over more short macros `\apOUT:0`, `\apOUT:1`, etc. Each such macro includes only 7 Digits pairs of the whole `\OUT`. The macro `\apOUTx` is invoked each 7 digit (the `\apnum0` register is decreased). It uses `\apnumL` value which is the `<num>` part of the next `\apOUT:<num>` control sequence. The `\apOUTx` defines this `<num>` as `\apOUTn` and initializes `\apOUT:<num>` as empty and adds the `<num>` to the list `\apOUTl`. When the creating of the next `\OUT` macro is definitely finished, the `\OUT` macro is assembled from the parts `\apOUT:0`, `\apOUT:1` etc. by the macro `\apOUTs <list of numbers><dot><comma>`.

apnum.tex

```
681: \def\apOUTx{\apnum0=7
682:   \edef\apOUTn{\the\apnumL}\edef\apOUTl{\apOUTl\apOUTn,}%
683:   \expandafter\def\csname apOUT:\apOUTn\endcsname{}%
684:   \advance\apnumL by1
685: }
686: \def\apOUTs#1,{\ifx.#1\else\csname apOUT:#1\expandafter\endcsname\expandafter\apOUTs\fi}
```

If a “function-like” macro needs a local counters then it is recommended to enclose all calculation into a group `\apINIT ... \apEND`. The `\apINIT` opens the group and prepares a short name `\do` and the macro `\localcounts<counters>;`. The typical usage is:

```
\def\MACRO#1{\relax \apINIT % function-like macro, \apINIT
  \evaldef\foo{#1}%          % preparing the parameter
  \localounts \N \M \K ;%    % local \newcount\N \newcount\M \newcount\K
  ...                        % calculation
  \apEND                     % end of \apINIT group
}
```

Note that `\localcounts` is used after preparing the parameter using `\evaldef` in order to avoid name conflict of local declared “variables” and “variables” used in `#1` by user.

The `\apINIT` sets locally `\localcounts` to be equivalent to `\apCOUNTS`. This macro increases the top index of allocated counters `\count10` (used in plain `TEX`) locally and declares the counters locally. It means that if the group is closed then the counters are deallocated and top index of counters `\count10` is returned to its original value.

apnum.tex

```
688: \def\apINIT{\begingroup \let\do=\apEVALxdo \let\localcounts=\apCOUNTS}
689: \def\apCOUNTS#1{\ifx.#1\else
690:   \advance\count10 by1 \countdef#1=\count10
691:   \expandafter\apCOUNTS\fi
692: }
```

The macro `\do <sequence>=<calculation>;` allows to write the calculation of Polish expressions more synoptic:

```
\do \X=\apPLUS{2}{\the\N};%    % is equivalent to:
\apPLUS{2}{\the\N}\let\X=\OUT
```

The `\do` macro is locally set to be equivalent to `\apEVALxdo`.

apnum.tex

```
693: \def\apEVALxdo#1=#2;{#2\let#1=\OUT}
```

The `\apRETURN` macro must be followed by `\fi`. It skips the rest of the block `\apINIT... \apEND` typically used in “function-like” macros. The `\apERR {<text>}` macro writes `<text>` as error message and

```
\apOUTx: 20–21, 27, 36    \apOUTn: 20–21, 27, 36    \apOUTl: 20, 26–27, 30, 36    \apOUTs: 20, 27, 36
\apINIT: 36–41, 44–47    \localcounts: 36–41, 44–45, 47    \apCOUNTS: 36    \do: 36, 38, 40–43, 45–47
\apEVALxdo: 36    \apRETURN: 37–41, 46–47    \apERR: 24, 29, 37–39, 41, 46–47
```

returns the processing of the block enclosed by `\apINIT... \apEND`. User can redefine it if the `\errmessage` isn't required.

```
695: \def\apRETURN#1\apEND{\fi\apEND}
696: \def\apERR#1{\errmessage{#1}}
```

apnum.tex

The `\apNOPT` macro removes the pt letters after expansion of $\langle dimen \rangle$ register. This is usable when we do a classical $\langle dimen \rangle$ calculation, see TBN page 80. Usage: `\expandafter\apNOPT\the\langle dimen \rangle`.

```
698: {\lccode'\?= '\p \lccode'\!= '\t \lowercase{\gdef\apNOPT#1!{\fi\apNOPT#1!}}
```

apnum.tex

The `\loop` macro from plain T_EX is redefined here in more convenient way. It does the same as original `\loop` by D. Knuth but moreover, it allows the construction `\if... \else... \repeat`.

```
700: \def\loop#1\repeat{\def\body{#1\relax\expandafter\body\fi}\body}
```

apnum.tex

2.10 Function-like Macros

The implementation of function-like macros `\ABS`, `\SGN`, `\iDIV`, `\iMOD`, `\iFLOOR`, `\iFRAC` are simple.

```
704: \def\ABS#1{\relax          % mandatory \relax for "function-like" macros
705:   \evaldef\OUT{#1}%        % evaluation of the input parameter
706:   \ifnum\apSIGN<0          % if (input < 0)
707:     \apSIGN=1              % sign = 1
708:     \apREMfirst\OUT        % remove first "minus" from OUT
709:   \fi                      % fi
710: }
711: \def\SGN#1{\relax \evaldef\OUT{#1}\edef\OUT{\the\apSIGN}\apE=0 }
712: \def\iDIV#1#2{\relax \apINIT          % calculation in group
713:   \evaldef\apAparam{#1}\apEadd\apAparam
714:   \evaldef\apBparam{#2}\apEadd\apBparam % evaluation of the parameters
715:   \apTOT=0 \apFRAC=0 \apDIV\apAparam\apBparam % integer division
716:   \apEND                          % end of group
717: }
718: \def\iMOD#1#2{\relax \apINIT          % calculation in group
719:   \evaldef\apAparam{#1}\apEadd\apAparam
720:   \evaldef\apBparam{#2}\apEadd\apBparam % evaluation of the parameters
721:   \apTOT=0 \apFRAC=0 \apDIV\apAparam\apBparam % integer division
722:   \let\OUT=\XOUT                % remainder is the output
723:   \apEND                          % end of group
724: }
725: \def\iFLOOR#1{\relax \evaldef\OUT{#1}\apEnum\OUT \apROUND\OUT0%
726:   \ifnum\apSIGN<0 \ifx\XOUT\empty \else \apPLUS\OUT{-1}\fi\fi
727:   \def\tmp{0}\ifx\tmp\OUT \apSIGN=0 \fi
728: }
729: \def\iFRAC#1{\relax
730:   \evaldef\OUT{#1}\apEnum\OUT \apROUND\OUT0% % preparing the parameter
731:   \ifx\XOUT\empty \def\OUT{0}\apSIGN=0        % empty fraction part means zero
732:   \else \ifnum\apSIGN<0
733:     \edef\XOUT{-.\XOUT}\apPLUS1\XOUT % OUT = 1 - .\XOUT
734:     \else \edef\OUT{.\XOUT}\apSIGN=1 % else OUT = .\XOUT
735:   \fi \fi
736: }
```

apnum.tex

The `\FAC` macro for **factorial** doesn't use recursive call because the T_EX group is opened in such case and the number of levels of T_EX group is limited (to 255 in my computer). But we want to calculate more factorial than only 255!

```
738: \def\FAC#1{\relax \apINIT          % "function-like" in the group, FAC = factorial
739:   \evaldef\OUT{#1}\apEnum\OUT        % preparing the parameter
740:   \localcounts \N;%                  % local \newcount
741:   \ifnum\apSIGN<0 \apERR{\string\FAC: argument {\OUT} cannot be negative}\apRETURN\fi
```

apnum.tex

`\apNOPT`: 37, 40, 42 `\loop`: 37–40, 42, 44–45, 47 `\ABS`: 3, 5, 37, 47 `\SGN`: 3, 37, 48
`\iDIV`: 3, 37, 48 `\iMOD`: 3, 37, 48 `\iFLOOR`: 3, 37, 48 `\iFRAC`: 3, 37, 48 `\FAC`: 3, 37–38, 47, 49


```

742: \let\tmp=\OUT \apROUND\tmp0% % test, if parameter is integer
743: \ifx\XOUT\empty \else \apERR{\string\FAC: argument {\OUT} must be integer}\apRETURN\fi
744: \N=\OUT\relax % N = param (error here if it is an big integer)
745: \ifnum\N=0\def\OUT{1}\apSIGN=1 \fi % special definition for factorial(0)
746: \loop \ifnum \N>2 \advance\N by-1 % loop if (N>2) N--
747: \apMUL{\OUT}{\the\N}\repeat % OUT = OUT * N , repeat
748: \apEND % end of group
749: }

```

The `\BINOM` $\{a\}\{b\}$ is **binomial coefficient** defined by

$$\binom{a}{b} = \frac{a!}{b!(a-b)!} = \frac{a(a-1)\cdots(a-b+1)}{b!} \quad \text{for integer } b > 0, \quad \binom{a}{0} = 1.$$

We use the formula where $(a-b)!$ is missing in numerator and denominator (second fraction) because of time optimization. Second advantage of such formula is that a need not to be integer. That is the reason why the `\BINOM` isn't defined simply as

```
\def\BINOM#1#2{\relax \evaldef{ \FAC{#1} / (\FAC{#2} * \FAC{(#1)-(#2)} )}}
```

The macro `\BINOM` checks if a is integer. If it is true then we choose $\backslash C$ as minimum of b and $a-b$. Then we calculate factorial of $\backslash C$ in the denominator of the formula (second fraction). And nominator includes $\backslash C$ factors. If a is non-negative integer and $a < b$ then the result is zero because one zero occurs between the factors in the nominator. Thus we give the result zero and we skip the rest of calculation. If a is non-integer, then $\backslash C$ must be b . The `\step` macro (it generates the factors in the nominator) is prepared in two versions: for a integer we use `\advance\A by-1` which is much faster than `\apPLUS\paramA{-1}` used for a non-integer.

apnum.tex

```

750: \def\BINOM#1#2{\relax \apINIT % BINOM = {#1 \choose #2} ...
751: \evaldef\apAparam{#1}\apEnum\apAparam
752: \evaldef\apBparam{#2}\apEnum\apBparam % preparation of the parameters
753: \localcounts \A \B \C ;% % local \newcounts
754: \let\OUT=\apBparam \apROUND\OUT0% % test if B is integer
755: \ifx\XOUT\empty\else\apERR{\string\BINOM: second arg. {\apBparam} must be integer}\apRETURN\fi
756: \let\OUT=\apAparam \apROUND\OUT0% % test if A is integer
757: \ifx\XOUT\empty % A is integer:
758: \A=\apAparam \B=\apBparam % A = #1, B = #2
759: \C=\A \advance\C by-\B % C = A - B
760: \ifnum\C>\B \C=\B \fi % if (C > B) C = B fi
761: \ifnum\A<0 \C=\B % if (A < 0) C = B fi
762: \else \ifnum\A<\B \def\OUT{0}\apSIGN=0 % if (0 <= A < B) OUT = 0 return
763: \expandafter\expandafter\expandafter \apRETURN \fi\fi
764: \def\step{\advance\A by-1 \apMUL\OUT{\the\A}}%
765: \else \C=\apBparam % A is not integer
766: \def\step{\let\apBparam\OUT \do\apAparam=\apPLUS\apAparam{-1}};%
767: \let\OUT=\apBparam \apMUL\OUT\apAparam}%
768: \fi
769: \ifnum\C=0 \def\OUT{1}\apSIGN=1 \apRETURN\fi
770: \do\D=\FAC{\the\C};% % D = C!
771: \let\OUT=\apAparam % OUT = #1
772: \loop \advance\C by-1 % loop C--
773: \ifnum\C>0 \step \repeat % if (C > 0) A--, OUT = OUT * A, repeat
774: \apDIV{\OUT}{\D}% % OUT = OUT / D
775: \apEND
776: }

```

The **square root** is computed in the macro `\SQRT` $\{a\}$ using Newton's approximation method. This method solves the equation $f(x) = 0$ (in this case $x^2 - a = 0$) by following way. Guess the initial value of the result x_0 . Create tangent to the graph of f in the point $[x_0, f(x_0)]$ using the knowledge about $f'(x_0)$ value. The intersection of this line with the axis x is the new approximation of the result x_1 . Do

`\BINOM`: 3, 38, 47 `\SQRT`: 3, 4, 39, 44, 47

the same with x_1 and find x_2 , etc. If you apply the general Newton method to the problem $x^2 - a = 0$ then you get the formula

$$\text{choose } x_0 \text{ as an initial guess, iterate: } x_{n+1} = \frac{1}{2} \left(x_n + \frac{a}{x_n} \right)$$

If $|x_{n+1} - x_n|$ is sufficiently small we stop the processing. In practice, we stop the processing, if the `\OUT` representation of x_{n+1} rounded to the `\apFRAC` is the same as the previous representation of x_n , i. e. `\ifx\Xn\OUT` in \TeX language. Amazingly, we need only about four iterations for 20-digits precision and about seven iterations for 50-digits precision, if the initial guess is good chosen.

The rest of the work in the `\SQRT` macro is about the right choose of the initial guess (using `\apSQRTTr` macro) and about shifting the decimal point in order to set the a value into the interval $[1, 100)$. The decimal point is shifted by $-M$ value. After calculation is done, the decimal point is shifted by $M/2$ value back. If user know good initial value then he/she can set it to `\apSQRTxo` macro. The calculation of initial value x_0 is skipped in such case.

```

777: \def\SQRT#1{\relax \apINIT           % OUT = Sqrt(#1) ...
778:   \evaldef\A{#1}%                   % parameter preparation
779:   \localcounts \M \E ;%              % local counters
780:   \E=\apE \apE=0
781:   \ifnum\apSIGN=0 \apRETURN\fi       % Sqrt(0) = 0 (OUT is set to 0 by previous \evaldef)
782:   \ifnum\apSIGN<0 \apERR{\string\SQRT: argument {\A} is out of range}\apRETURN\fi
783:   \ifodd\E \apROLL\A{-1}\advance\E by1 \fi % we need the E representation with even exponent
784:   \let\B=\A \let\C=\A
785:   \apDIG\C\relax \M=\apnumD          % M is the number of digits before decimal point
786:   \advance\M by-2 \ifodd\M \advance\M by1 \fi % M = M - 2 , M must be even
787:   \ifx\apSQRTxo\undefined           % we need to calculate X0
788:     \ifnum\M=0 \else \apROLL\B{-\M}\divide\M by2 \fi % shift decimal point by -M, M = M / 2
789:     \apSQRTTr\B \let\Xn=\OUT          % Xn = estimate of Sqrt
790:     \ifnum\M<0 \let\A=\B \fi          % if (A < 1) calculate with B where decimal point is shifted
791:     \ifnum\M>0 \apROLL\Xn \M \fi      % if (A >= 100) shift the decimal point of initial guess
792:   \else \let\Xn=\apSQRTxo \fi
793:   \loop                               % loop ... Newton's method
794:     \apDIV{\apPLUS{\Xn}{\apDIV{\A}{\Xn}}}{2}% % OUT = (Xn + A/Xn) / 2
795:     \ifx\OUT\Xn \else                 % if (OUT != Xn)
796:       \let\Xn=\OUT \repeat            % Xn = OUT, repeat
797:     \ifnum\M<0 \apROLL\OUT\M \fi      % shift the decimal point by M back
798:     \apE=\E \divide\apE by2          % correct the E exponent
799:   \apEND
800: }
```

Note that if the input $a < 1$, then we start the Newton's method with b . It is the value a with shifted decimal point, $b \in [1, 100)$. On the other hand, if $a \geq 1$ then we start the Newton's method directly with a , because the second derivative $(x^2)''$ is constant so the speed of Newton's method is independent on the value of x . And we need to calculate the `\apFRAC` digits after the decimal point.

The macro `\apSQRTTr` $\langle number \rangle$ accepts $\langle number \rangle$ in the interval $[1, 100]$ and makes a roughly estimation of square root of the $\langle number \rangle$ in the `\OUT` macro. It uses only classical $\langle dimen \rangle$ calculation, it doesn't use any `apnum.tex` operations. The result is based on the linear approximation of the function $g(x) = \sqrt{x}$ with known exact points $[1, 1], [4, 2], [9, 3], \dots, [100, 10]$. Note, that the differences between x_i values of exact points are 3, 5, 7, \dots , 19. The inverted values of these differences are pre-calculated and inserted after `\apSQRTTr` macro call.

The `\apSQRTTr` macro operates repeatedly for $i = 1, \dots, 10$ until $\dimen0 = x < x_i$. Then the `\apSQRTTrb` is executed. We are in the situation $\dimen0 = x \in [x_{i-1}, x_i)$, $g(x_i) = i$, $g(x_{i-1}) = i - 1$ and the calculation of $\OUT = g(x_{i-1}) + (x - x_{i-1})/(x_i - x_{i-1})$ is performed. If $x \in [1, 4)$ then the linear approximation is worse. So, we calculate additional linear correction in $\dimen1$ using the pre-calculated value $\sqrt{2} - 1.33333 \doteq 0.08088$ here.

```

801: \def\apSQRTTr#1{\dimen0=#1pt \apnumB=1 \apnumC=1 \apSQRTTr}
802: \def\apSQRTTrb{\advance\apnumB by2 \advance\apnumC by\apnumB % B = difference, C = x_i
803:   \ifnum\apnumC>100 \def\OUT{10}\else
```

```

804: \ifdim\dimen0<\apnumC pt \apSQRTTrb \else
805: \expandafter\expandafter\expandafter\apSQRTTra\fi\fi
806: }
807: \def\apSQRTTrb{% x = dimen0, B = x_i - x_{i-1}, C = x_i = i
808: \ifdim\dimen0<4pt
809: \ifdim\dimen0>2pt \dimen1=4pt \advance\dimen1 by-\dimen0 \divide\dimen1 by2
810: \else \dimen1=\dimen0 \advance\dimen1 by-1pt \fi
811: \dimen1=.080884\dimen1 % dimen1 = additional linear correction
812: \else \dimen1=0pt \fi
813: \advance\apnumC by-\apnumB % C = x_{i-1}
814: \advance\dimen0 by-\apnumC pt % dimen0 = (x - x_{i-1})
815: \divide\dimen0 by\apnumB % dimen0 = (x - x_{i-1}) / difference
816: \divide\apnumB by2 % B = i-1 = g(x_{i-1})
817: \advance\dimen0 by\apnumB pt % dimen0 = g(x_{i-1}) + (x - x_{i-1}) / (x_i-x_{i-1})
818: \advance\dimen0 by\dimen1 % dimen0 += additional linear correction
819: \edef\OUT{\expandafter\apNOPT\the\dimen0}% OUT = dimen0
820: }

```

The exponential function e^x is implemented in the `\EXP` macro using Taylor series at zero point:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

If $x \in (0, 1)$ then this series converges relatively quickly.

The macro `\EXP` takes its argument. If it is negative, remember this fact, remove minus sign and do `\OUT=1/\OUT` in final step. Now, the argument is positive always. If the argument is greater than 1, do argument = argument/2 and increase K register. Do this step in the loop until argument < 1 . After `\OUT` is calculated then we do `\OUT=\OUT^2` in the loop K times, because $e^{2x} = (e^x)^2$. Now we are ready to calculate the exponential of positive argument which is less than 1. This is done using loop of Taylor series. `\S` is total sum, `\Sn` is the new addition in the n -th step. If `\Sn` is zero (in accordance to the `\apFRAC` register) then we stop the calculation.

apnum.tex

```

821: \def\EXP#1{\relax\apINIT % OUT = EXP(#1) ...
822: \evaldef\OUT{#1}\apEnum\OUT % OUT = #1
823: \localcounts \N \K;%
824: \ifnum\apSIGN=0 \def\OUT{1}\apSIGN=1 \apRETURN \fi
825: \edef\digits{\the\apFRAC}\advance\apFRAC by3
826: \edef\signX{\the\apSIGN}%
827: \ifnum\apSIGN<0 \apSIGN=1 \apREMfirst\OUT \fi % remove "minus" sign
828: \K=0 \N=0 % K = 0, N = 0
829: \def\testDot ##1##2\relax##3{\ifx##1.}%
830: \loop \expandafter \testDot\OUT \relax % loop if (OUT >= 1)
831: \iftrue \else % OUT = OUT/2
832: \apDIV\OUT{2}% % K++
833: \advance\K by1 % repeat
834: \repeat % now: #1 = 2^K * OUT, OUT < 1
835: \advance\apFRAC by\K
836: \def\S{1}\def\Sn{1}\N=0 \let\X=\OUT % S = 1, Sn = 1, N = 0, X = OUT
837: \loop \advance\N by1 % loop N++
838: \do\Sn=\apDIV{\apMUL\Sn\X}{\the\N};% % Sn = Sn * X / N
839: \apTAYLOR\iftrue \repeat % S = S + Sn (... Taylor)
840: \N=0
841: \loop \ifnum\N < \K % loop if (N < K)
842: \apPOW\OUT{2}\apROUND\OUT\apFRAC % OUT = OUT^2
843: \advance\N by1 \repeat % N++
844: \ifnum\signX<0 \apDIV 1\OUT \fi % if (signX < 0) OUT = 1 / OUT
845: \apROUND\OUT\digits \apSIGN=1 % EXP is always positive
846: \apEND
847: }

```

The macro `\apTAYLOR` is ready for general usage in the form:

```

\def\S{...}\def\Sn{...}\N=... % setting initial values for N=0
\loop

```

`\EXP`: 3, 4, 6, 40–41, 43, 47 `\apTAYLOR`: 40–42, 44–45, 47

```

... % auxiliary calculation
\do\Sn=\apDIV{...}{...};% % calculation of new addition \Sn
% (division must be the last activity)
\apTAYLOR \iftrue \repeat % does S = S + Sn and finishes if Sn = 0

```

apnum.tex

```
848: \def\apTAYLOR#1{\ifnum\apSIGN=0 \let\OUT=\S \else \apPLUS\S\Sn \let\S=\OUT }
```

The **logarithm function** $\ln x$ (inverse to e^x) is implemented in `\LN` macro by Taylor series in the point zero of the $\arg \tanh$ function:

$$\ln x = 2 \arg \tanh \frac{x-1}{x+1} = 2 \left(\frac{x-1}{x+1} + \frac{1}{3} \left(\frac{x-1}{x+1} \right)^3 + \frac{1}{5} \left(\frac{x-1}{x+1} \right)^5 + \dots \right).$$

This series converges quickly when x is approximately equal to one. The idea of the macro `\LN` includes the following steps:

- Whole calculation is in the group `\apINIT...\apEND`. Enlarge the `\apFRAC` numeric precision by three digits in this group.
- Read the argument `\X` using `\evaldef`.
- If the argument is non positive, print error and skip the next processing.
- If the argument is in the interval $(0, 1)$, set new argument as $1/\text{argument}$ and remember the “minus” sign for the calculated `\OUT`, else the `\OUT` remains to be positive. This uses the identity $\ln(1/x) = -\ln x$.
- shift the decimal point of the argument by M positions left in order to the new argument is in the interval $[1, 10)$.
- Let $x \in [1, 10)$ be the argument calculated as mentioned before. Calculate roughly estimated $\widetilde{\ln x}$ using `\apLNr` macro. This macro uses linear interpolation of the function $\ln x$ in eleven points in the interval $[1, 10]$.
- Calculate $A = x / \exp(\widetilde{\ln x})$. The result is approximately equal to one, because $\exp(\ln x) = x$.
- Calculate $\ln A$ using the Taylor series above.
- The result of $\ln x$ is equal to $\ln A + \widetilde{\ln x}$, because $x = A \cdot \exp(\widetilde{\ln x})$ and $\ln(ab) = \ln a + \ln b$.
- The real argument is in the form $x \cdot 10^M$, so `\OUT` is equal to $\ln x + M \cdot \ln(10)$ because $\ln(ab) = \ln a + \ln b$ and $\ln(10^M) = M \ln(10)$. The $\ln(10)$ value with desired precision is calculated by `\apLNtenexec` macro. This macro saves its result globally when firstly calculated and use the calculated result when the `\apLNtenexec` is called again.
- Round the `\OUT` to the `\apFRAC` digits.
- Append “minus” to the `\OUT` if the input argument was in the interval $(0, 1)$.

apnum.tex

```

850: \def\LN#1{\relax \apINIT % OUT = LN(#1) ...
851: \evaldef\X{#1}% % X = #1
852: \localcounts \M \N \E;%
853: \E=\apE
854: \def\round{\apROUND\OUT\apFRAC}%
855: \edef\digits{\the\apFRAC}\advance\apFRAC by4
856: \ifnum\apSIGN>0 \else \apERR{\string\LN: argument {\X} is out of range}\apRETURN\fi
857: \apDIG\OUT\relax \M=\apnumD % find M: X = mantissa * 10^M
858: \ifnum\M>-\E \def\sgnout{1}\else % if X in (0,1):
859: \def\sgnout{-1}% % sgnout = -1
860: \do\X=\apDIV 1\X;\E=-\E % X = 1/X
861: \apDIG\OUT\relax \M=\apnumD % find M: X = mantissa * 10^M
862: \fi % else sgnout = 1
863: \advance\M by-1 % M = M - 1
864: \ifnum\M=0 \else\apROLL\X{-\M}\fi % X = X * 10^(-M), now X in (1,10)
865: \advance\M by\E % M = M + E (scientific format of numbers)
866: \do\lnX=\apLNr\X;% % lnX = LN(X) ... roughly estimate
867: \do\A=\apDIV\X{\EXP\lnX};% % A = X / EXP(lnX) ... A=approx= 1
868: \apLNtaylor % OUT = LN(A)
869: \do\LNOUT=\apPLUS\OUT\lnX;% % LNOUT = OUT + LNrOUT

```

`\LN`: 3, 4, 6, 41–42, 47

```

870: \ifnum\M>0 % if M > 0
871: \apLNtenexec % LNtenOUT = ln(10)
872: \apPLUS\LNOUT{\apMUL{\the\M}{\apLNten}}% OUT = LNOUT + M * LNten
873: \fi
874: \ifnum\apSIGN=0 \else \apSIGN=\sgnout \fi % if (OUT != 0) apSIGN = saved sign
875: \apROUND\OUT\digits % round result to desired precision
876: \ifnum\apSIGN<0 \xdef\OUT{-\OUT}\else \global\let\OUT=\OUT \fi
877: \apEND
878: }

```

The macro `\apLNtaylor` calculates $\ln A$ for $A \approx 1$ using Taylor series mentioned above.

```

879: \def\apLNtaylor{%
880: \apDIV{\apPLUS{A}{-1}}{\apPLUS{A}{1}}% % OUT = (A-1) / (A+1)
881: \ifnum\apSIGN=0 \def\OUT{0}\else % ln 1 = 0 else:
882: \let\Sn=\OUT \let\Kn=\OUT \let\S=\OUT % Sn = OUT, Kn = OUT, S = OUT
883: \apPOW\OUT{2}\round \let\XX=\OUT % XX = OUT^2
884: \N=1 % N = 1
885: \loop \advance\N by2 % loop N = N + 2
886: \do\Kn=\apMUL\Kn\XX\round;% % Kn = Kn * XX
887: \do\Sn=\apDIV\Kn{\the\N};% % Sn = Kn / N
888: \apTAYLOR\iftrue \repeat % S = S + Sn (Taylor)
889: \apMUL\S{2}% % OUT = 2 * OUT
890: \fi
891: }

```

The macro `\apLNr` finds an estimation $\widetilde{\ln x}$ for $x \in [1, 10)$ using linear approximation of $\ln x$ function. Only direct `<dimen>` and `<count>` calculation with \TeX registers is used, no long numbers `apnum.tex` calculation. The $\ln x_i$ is pre-calculated for $x_i = i, i \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ and the values are inserted after the `\apLNra` macro call. The input value x is set as `\dimen0`.

The `\apLNra` `{<valueA>}{<valueB>}` macro reads the pre-calculated values repeatedly in the loop. The loop ends if `\apnumC` (i. e. x_i) is greater than x . Then we know that $x \in [x_{i-1}, x_i)$. The linear interpolation is

$$\widetilde{\ln x} = f(x_{i-1}) + (f(x_i) - f(x_{i-1})) (x - x_{i-1}),$$

where $f(x_{i-1}) = \langle \text{valueA} \rangle$, $f(x_i) = \langle \text{valueB} \rangle$ and $x = \text{\dimen0}$. The rest of the pre-calculated values is skipped by processing `\next` to `\relax`.

The pre-calculated approximation of $\ln 10$ is saved in the macro `\apLNrtten` because we use it at more places in the code.

```

892: \def\apLNr#1{\dimen0=#1pt \apnumC=1
893: \apLNra {0}{.69}{1.098}{1.386}{1.609}{1.791}{1.9459}{2.079}{2.197}{\apLNrtten}\relax
894: }
895: \def\apLNra #1#2{\advance\apnumC by1
896: \ifx\relax#2\relax \let\OUT=\apLNrtten \let\apNext=\relax
897: \else
898: \ifdim\dimen0<\apnumC pt % linear interpolation:
899: \advance\dimen0 by-\apnumC pt \advance\dimen0 by1pt % dimen0 = x - x_{i-1}
900: \dimen1=#2pt \advance\dimen1 by-#1pt % dimen1 = f(x_i) - f(x_{i-1})
901: \dimen1=\expandafter\apNOPT\the\dimen0 \dimen1 % dimen1 = (x - x_{i-1}) * dimen1
902: \advance\dimen1 by#1pt % dimen1 = f(x_{i-1}) + dimen1
903: \edef\OUT{\expandafter\apNOPT\the\dimen1}% % OUT = dimen1
904: \def\apNext#1\relax{}%
905: \else \def\apNext{\apLNra{#2}}%
906: \fi\fi \apNext
907: }
908: \def\apLNrtten{2.302585} % apLNrtten = ln 10 (roughly)

```

The `\apLNtenexec` macro calculates the $\ln 10$ value with the precision given by `\apFRAC`. The output is prepared to the `\apLNten` macro. The `\apLNtenexec` saves globally the result to the macro `\LNten`: `<apFRAC>` in order to use it if the value is needed again. This saves time.

`\apLNtaylor`: 41–43 `\apLNr`: 41–42 `\apLNra`: 42 `\apLNrtten`: 42–43 `\apLNtenexec`: 41–43
`\apLNten`: 42–43

```

909: \def\apLNtenexec{%                                % OUT = ln 10 ...
910:   \expandafter\ifx\csname LNten:\the\apFRAC\endcsname \relax
911:     \begingroup \apTOT=0
912:     \do{A=\apDIV{10}{\EXP\apLNrten}};%           % A = 10 / exp(LNrten)
913:     \apLNtaylor                                   % OUT = ln A
914:     \apPLUS\OUT\apLNrten                           % OUT = OUT + LNrten
915:     \global\expandafter\let\csname LNten:\the\apFRAC\endcsname=\OUT
916:   \endgroup
917: \fi
918: \expandafter\let\expandafter \apLNten \csname LNten:\the\apFRAC\endcsname
919: }

```

The constant π is saved in the `\apPIvalue` macro initially with 30 digits. If user needs more digits (using `\apFRAC > 30`) then the `\apPIvalue` is recalculated and the `\apPIDigits` is changed appropriately.

```

920: \def\apPIvalue{3.141592653589793238462643383279}
921: \def\apPIDigits{30}

```

The macro `\apPIexec` prepares the π constant with `\apFRAC` digits and saves it to the `\apPI` macro. And $\pi/2$ constant with `\apFRAC` digits is saved to the `\apPIhalf` macro. The `\apPIexec` uses macros `\apPI:\langle apFRAC \rangle` and `\apPIh:\langle apFRAC \rangle` where desired values are usually stored. If the values are not prepared here then the macro `\apPIexecA` calculates them.

```

922: \def\apPIexec{%
923:   \expandafter\ifx\csname apPI:\the\apFRAC\endcsname \relax \apPIexecA \else
924:     \expandafter\let\expandafter\apPI\csname apPI:\the\apFRAC\endcsname
925:     \expandafter\let\expandafter\apPIhalf\csname apPIh:\the\apFRAC\endcsname
926:   \fi
927: }

```

The macro `\apPIexecA` creates the π value with `\apFRAC` digits using the `\apPIvalue`, which is rounded if `\apFRAC < \apPIDigits`. The `\apPIhalf` is calculated from `\apPI`. Finally the macros `\apPI:\langle apFRAC \rangle` and `\apPIh:\langle apFRAC \rangle` are saved globally for saving time when we need such values again.

```

928: \def\apPIexecA{%
929:   \ifnum\apPIDigits<\apFRAC \apPIexecB \fi
930:   \let\apPI=\apPIvalue
931:   \ifnum\apPIDigits>\apFRAC \apROUND\apPI\apFRAC \fi
932:   \apnumP=\apTOT \apTOT=0 \apDIV\apPI2\let\apPIhalf=\OUT \apTOT=\apnumP
933:   \global\expandafter\let\csname apPI:\the\apFRAC\endcsname=\apPI
934:   \global\expandafter\let\csname apPIh:\the\apFRAC\endcsname=\apPIhalf
935: }

```

If `\apFRAC > \apPIDigits` then new `\apPIvalue` with desired decimal digits is generated using `\apPIexecB` macro. The Chudnovsky formula is used:

$$\pi = \frac{53360 \cdot \sqrt{640320}}{S}, \quad S = \sum_{n=0}^{\infty} \frac{(6n)! (13591409 + 545140134n)}{(3n)! (n!)^3 (-262537412640768000)^n}$$

This converges very good with 14 new calculated digits per one step where new S_n is calculated. Moreover, we use the identity:

$$F_n = \frac{(6n)!}{(3n)! (n!)^3 (-262537412640768000)^n}, \quad F_n = F_{n-1} \cdot \frac{8(6n-1)(6n-3)(6n-5)}{n^3 (-262537412640768000)}$$

`\apPIvalue:` 43–44 `\apPIDigits:` 43–44 `\apPIexec:` 43–47 `\apPI:` 43–45 `\apPIhalf:` 43–47
`\apPIexecA:` 43 `\apPIexecB:` 43–44

and we use auxiliary integer constants A_n, B_n, C_n with following properties:

$$\begin{aligned} A_0 &= B_0 = C_0 = 1, \\ A_n &= A_{n-1} \cdot 8(6n-1)(6n-3)(6n-5), \quad B_n = B_{n-1} \cdot n^3, \quad C_n = C_{n-1} \cdot (-262537412640768000), \\ F_n &= \frac{A_n}{B_n C_n}, \\ S_n &= \frac{A_n(13591409 + 545140134n)}{B_n C_n} \end{aligned}$$

```

936: \def\apPiexecB{\apINIT
937:   \localcounts \N \a \c;%
938:   \apTOT=0 \advance\apFRAC by2
939:   \def\apSQRTxo{800.199975006248}% initial value for Newton method for SQRT
940:   \SQRT{640320}%
941:   \let\sqrtval=\OUT
942:   \N=0 \def\An{1}\def\Bn{1}\def\Cn{1}\def\S{13591409}%
943:   \loop
944:     \advance\N by 1
945:     \a=\N \multiply\N by6 \advance\N by-1 \c=\a
946:     \advance\N by-2 \multiply\N by\N % An = An * 8 * (6N-5) *
947:     \advance\N by-2 \multiply\N by8 %      * (6N-3) * (6N-1)
948:     \apMUL\An{\apMUL{\the\N}{\the\N}}\let\An=\OUT
949:     \c=\N \multiply\N by\N % Bn = Bn * n^3
950:     \apMUL\Bn{\apMUL{\the\N}{\the\N}}\let\Bn=\OUT
951:     \apMUL\Cn{-262537412640768000}\let\Cn=\OUT % Cn = Cn * K3
952:     \apDIV{\apMUL\An{\apPLUS{13591409}{\apMUL{545140134}{\the\N}}}}{\apMUL\Bn\Cn}%
953:     \let\S=\OUT % Sn = An * (K1 + K2 * N) / (Bn * Cn)
954:     \apTAYLOR \iftrue \repeat
955:   \advance\apFRAC by-2
956:   \apDIV{\apMUL{\sqrtval}{53360}}\S
957:   \global\let\apPivalue=\OUT
958:   \xdef\apPidigits{\the\apFRAC}%
959:   \apEND
960: }

```

The macros for users `\PI` and `\PIhalf` are implemented as “function-like” macros without parameters.

```

961: \def\PI{\relax \apPiexec \let\OUT=\apPI}
962: \def\PIhalf{\relax \apPiexec \let\OUT=\apPIhalf}

```

The macros `\SIN` and `\COS` use the Taylor series

$$\begin{aligned} \sin x &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots \\ \cos x &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots \end{aligned}$$

These series converge good for $|x| < 1$. The main problem is to shift the given argument $x \in \mathbf{R}$ to the range $[0, 1)$ before the calculation of the series is started. This task is done by `\apSINCOSa` macro, the common code for both, `\SIN` and `\COS` macros.

The macro `\apSINCOSa` does the following steps:

- It advances `\apFRAC` by three and evaluates the argument.
- Note, that the macro `\apSINCOSx` means `\apSINx` or `\apCOSx` depending on the given task.
- The macro `\signK` includes 1. It can be recalculated to -1 later.
- If the argument is zero then the result is set and next computation is skipped. This test is processed by `\apSINCOSo\apCOSx`.

`\PI`: 3, 4, 44, 47 `\PIhalf`: 3, 4, 44, 47 `\SIN`: 3, 4, 8, 44–49 `\COS`: 3, 4, 44–49
`\apSINCOSa`: 44–46

- If the argument is negative then remove minus and save `\sign`. This `\sign` will be applied to the result. The `\sign` is always + when `\COS` is calculated. This follows the identities $\sin(-x) = -\sin x$ and $\cos(-x) = \cos x$.
- The `\apFRAC` is saved and `\apTOT=0`.
- The `\apPIexec` is processed. The `\apPI` and `\apPIhalf` are ready after such processing.
- After `\X div \apPI` (rounded to integer) we have `\K` in `\OUT`, where $X = x' + K \cdot \pi$ and $x' \in [0, \pi)$. We set $X := x'$ because of the identities $\sin x = (-1)^k \sin(x + k\pi)$, $\cos x = (-1)^k \cos(x + k\pi)$. The sign $(-1)^k$ is saved to `\signK` macro.
- If the x' is zero then the result is set by `\apSINCOSo\apCOSx` and the rest of calculating is skipped.
- The $|X - \pi/2|$ is saved to `\XmPIh` macro.
- If $X \in (\pi/4, \pi/2)$ then $x' = \text{\XmPIh}$. We use identities $\sin x = \cos(\pi/2 - x)$, $\cos x = \sin(\pi/2 - x)$. Set $X = x'$. The meaning of `\apSINCOSx` (`\apSINx` or `\apCOSx`) is flipped in such case.
- If the x' is zero then the result is set by `\apSINCOSo\apSINx` and the rest of calculating is skipped.
- Now $X \in (0, \pi/4)$, i. e. $|X| < 1$ and we can use Taylor series. The `\apSINCOSx` (i. e. `\apSINx` or `\apCOSx`) macro initializes the computation of Taylor series mentioned above. The $XX = X^2$ is prepared. The Taylor series is processed in the loop as usually.
- The the sign of the output is `\sign\signK`.
- If the sign of the result is negative, the “minus” is added to the `\OUT`.

apnum.tex

```

964: \def\SIN{\relax \let\apSINCOSx=\apSINx \apSINCOSa}
965: \def\COS{\relax \let\apSINCOSx=\apCOSx \apSINCOSa}
966: \def\apSINCOSa#1{\apINIT
967:   \advance\apFRAC by3
968:   \evaldef\X{#1}\apEnum\X
969:   \def\signK{1}\apSINCOSo\apCOSx
970:   \ifnum\apSIGN<0 \apREMfirst\X \def\sign{-}\else\def\sign{+}\fi
971:   \ifx\apSINCOSx\apCOSx \def\sign{+}\fi
972:   \edef\apFRACsave{\the\apFRAC}%
973:   \apPIexec
974:   \apFRAC=0 \apDIV\X\apPI           % OUT = X div PI
975:   \ifnum\apSIGN=0 \apSIGN=1 \else
976:     \let\K=\OUT
977:     \do\X=\apPLUS\X{-\apMUL\K\apPI};% X := X - K * PI
978:     \apROLL\K{-1}\apROUND\K{0}%
979:     \ifodd 0\XOUT\space \def\signK{-1}\else\def\signK{1}\fi
980:   \fi
981:   \apSINCOSo\apCOSx
982:   \apFRAC=\apFRACsave \relax
983:   \do\XmPIh=\apPLUS\X{-\apPIhalf};% XmPIh = | X - PI/2 |
984:   \apSINCOSo\apSINx
985:   \ifnum\apSIGN<0 \apREMfirst\XmPIh
986:   \else % X in (PI/2, PI)
987:     \do\X=\apPLUS\apPI{-\X};%
988:     \ifx\apSINCOSx\apCOSx \apSIGN=-\signK \edef\signK{\the\apSIGN}\fi
989:   \fi % X in (0, PI/2):
990:   \apMINUS\X{.78}% % OUT = X - cca PI/4
991:   \ifnum\apSIGN<0 \else % if X in (PI/4, PI/2) :
992:     \let\X=\XmPIh % X = | X - PI/2 |; SIN <-> COS
993:     \ifx\apSINCOSx\apSINx \let\apSINCOSx=\apCOSx \else \let\apSINCOSx=\apSINx \fi
994:   \fi
995:   \localcounts \N \NN;%
996:   \do\XX=\apPOW\X{2}\ROUND\OUT\apFRAC;%
997:   \apSINCOSx % X in (0, PI/4), initialize Taylor SIN X or COS X
998:   \loop
999:     \advance\N by1 \NN=\N
1000:     \advance\N by1 \multiply\NN by\N
1001:     \do\Sn=\apDIV{\apMUL\Sn\XX}{-\the\NN};% Sn = - Sn * X^2 / N*(N+1)
1002:     \apTAYLOR \iftrue\repeat
1003:   \apSIGN=\sign\signK
1004:   \ifnum\apTOT=0 \advance\apFRAC by-3 \else \apFRAC=\apTOT \fi
1005:   \ifnum\apFRAC<0 \apFRAC=-\apFRAC \fi
1006:   \apROUND\OUT\apFRAC
1007:   \def\X{0}\ifx\OUT\X \apSIGN=0 \fi
1008:   \ifnum\apSIGN<0 \edef\OUT{-\OUT}\fi

```

```
1009: \apEND
1010: }
```

The macros `\apSINx` and `\apCOSx` initialize the calculation of the Taylor series.

apnum.tex

```
1011: \def\apSINx{\let\S=X \N=1 \let\Sn=X}
1012: \def\apCOSx{\def\S{1}\N=0 \let\Sn=S}
```

The `\apSINCOSo` *(sequence)* macro is used three times in the `\apSINCOSA`. It tests if the current result is zero. If it is true then the `\OUT` is set as zero or it is set to `\signK` (if processed function is equal to the *(sequence)*).

apnum.tex

```
1013: \def\apSINCOSo#1{\ifnum\apSIGN=0 \ifx#1\SCgo \apSIGN=\signK \let\OUT=\signK \fi \apRETURN\fi}
```

The macro `\TAN` uses the identity $\tan x = \sin x / \cos x$ and calculates the denominator first. If it is zero then `\apERR` prints “out of range” message else the result is calculated.

apnum.tex

```
1014: \def\TAN#1{\relax \apINIT
1015: \advance\apFRAC by3
1016: \evaldef\X{#1}\apEnum\X
1017: \advance\apFRAC by-3
1018: \do\denom=\COS\X;%
1019: \ifnum\apSIGN=0 \apERR{\string\TAN: argument {\X} is out of range}\apRETURN\fi
1020: \SIN\X\message{\OUT/\denom}%
1021: \apDIV{\SIN\X}\denom
1022: \apEND
1023: }
```

The macro `\ATAN` calculates the inverse of tangens using series

$$\arctan \frac{1}{x} = \frac{x}{1+x^2} + \frac{2}{3} \frac{x}{(1+x^2)^2} + \frac{2}{3} \frac{4}{5} \frac{x}{(1+x^2)^3} + \frac{2}{3} \frac{4}{5} \frac{6}{7} \frac{x}{(1+x^2)^4} + \dots$$

This converges relatively good for $|x| > 1$. I was inspired by the Claudio Kozický’s semestral work from the course “Typography and \TeX ” at ČVUT in Prague.

The macro `\ATAN` takes the argument x and uses identity $\arctan(-x) = -\arctan(x)$ when x is negative. If $x > 1$ then the identity

$$\arctan(x) = \frac{\pi}{2} - \arctan \frac{1}{x}$$

is used and $\arctan(1/x)$ is calculated by `\apATANox` macro using the series above. Else the argument is re-calculated $x := 1/x$ and the `\apATANox` is used. When $x = 1$ then the `\apPIhalf/2` is returned directly.

apnum.tex

```
1024: \def\ATAN#1{\relax \apINIT
1025: \advance\apFRAC by3
1026: \evaldef\X{#1}\apEnum
1027: \ifnum\apSIGN=0 \def\OUT{0}\apRETURN\fi
1028: \ifnum\apSIGN<0 \def\sign{-}\apREMfirst\X \else\def\sign{}\fi
1029: \let\tmp=X \apDIG\tmp\relax
1030: \ifnum\apnumD>0 % if X > 1:
1031: \apPiexec % OUT = apPIhalf - apATANox
1032: \def\tmp{1}\ifx\tmp\X \apDIV\apPIhalf2\else \apATANox \apPLUS\apPIhalf{-\OUT}\fi
1033: \else % else
1034: \do\X=\apDIV{1}\X;% X := 1/X
1035: \apATANox % OUT = apATANox
1036: \fi
1037: \ifnum\apTOT=0 \advance\apFRAC by-3 \else \apFRAC=\apTOT \fi
1038: \ifnum\apFRAC<0 \apFRAC=-\apFRAC \fi
1039: \apROUND\OUT\apFRAC
1040: \ifx\sign\empty\apSIGN=1 \else \edef\OUT{-\OUT}\apSIGN=-1 \fi
1041: \apEND
1042: }
```

`\apSINx`: 44–46 `\apCOSx`: 44–46 `\apSINCOSo`: 44–46 `\TAN`: 3, 4, 46–47 `\ATAN`: 3, 4, 46–47

The macro `\apATANox` calculates $\arctan(1/x)$ using series mentioned above.

apnum.tex

```

1043: \def\apATANox{%
1044:   \localcount \N;%
1045:   \do\XX=\apPLUS{1}{\apPOW{X{2}}}\apROUND\OUT\apFRAC;% XX = 1 + X^2
1046:   \do\Sn=\apDIV{X\XX}\apROUND\OUT\apFRAC;%           % Sn = X / (1+X^2)
1047:   \N=1 \let\S=\Sn
1048:   \loop
1049:     \advance\N by1
1050:     \do\Sn=\apMUL{the\N}\Sn;%
1051:     \advance\N by1
1052:     \do\Sn=\apDIV\Sn{\apMUL{the\N}\XX};% Sn = Sn * N / ((N+1) * (1+X^2))
1053:     \apTAYLOR \iftrue \repeat
1054: }

```

The macros `\ASIN` and `\ACOS` for functions $\arcsin(x)$ and $\arccos(x)$ are implemented using following identities:

$$\arcsin(x) = \arctan \frac{x}{\sqrt{1-x^2}}, \quad \arccos(x) = \frac{\pi}{2} - \arcsin(x)$$

apnum.tex

```

1055: \def\ASIN#1{\relax \apINIT
1056:   \evaldef{X{#1}}\apEnum{X} \edef\sign{\the\apSIGN}%
1057:   \apPLUS 1{-\apPOW{X2}}% OUT = 1 - X^2
1058:   \ifnum\apSIGN<0 \apERR{\string\ASIN: argument {X} is out of range}\apRETURN\fi
1059:   \do\sqr=\SQRT\OUT;%      sqrt = SQRT {1 - X^1}
1060:   \ifnum\apSIGN=0 \apPiexec
1061:     \ifnum\sign<0 \edef\OUT{-\apPIhalf}\apSIGN=-1 % ASIN(-1) = -PI/2
1062:     \else \let\OUT=\apPIhalf \apSIGN=1 \fi % ASIN(1) = PI/2
1063:   \apRETURN \fi
1064:   \ATAN{X/\sqr}%          OUT = arctan ( X / SQRT {1 - X^2} )
1065:   \apEND
1066: }
1067: \def\ACOS#1{\relax \apPiexec \apPLUS\apPIhalf{-\ASIN{#1}}}

```

2.11 Printing expressions

The `\eprint {⟨expression⟩}{⟨declaration⟩}` macro works in the group `\bgroup... \egroup`. This means that the result in math mode is math-Ord atom. The macro interprets the `⟨expression⟩` in the first step like `\evaldef`. This is done by `\apEVALb#1\limits`. The result is stored in the `\tmpb` macro in Polish notation. Then the internal initialization is processed in `\apEPi` and user-space initialization is added in #2. Then `\tmpb` is processed. The `\apEpe` can do something end-game play but typically it is `\relax`.

apnum.tex

```

1071: \def\eprint#1#2{\bgroup \apnumA=0 \apnumE=1 \apEVALb#1\limits
1072:   \let\apEpe=\relax \apEPi #2\tmpb \apEpe \egroup
1073: }

```

The `\apEPi` macro replaces the meaning of all macros typically used in Polish notation of the expression. The original meaning is “to evaluate”, the new meaning is “to print”. The macro `\apEPi` is set to `\relax` in the working group because nested `⟨expressions⟩` processed by nested `\eprints` need not to be initialized again.

There is second initialization macro `\apEPj` (similar to the `\apEPi`) which is empty by default. Users can define their own function-like functions and they can put the printing initialization of such macros here.

apnum.tex

```

1074: \def\apEPi{\let\apPLUS=\apEPplus \let\apMINUS=\apEPminus
1075:   \let\apMUL=\apEPmul \let\apDIV=\apEPdiv \let\apPOWx=\apEPpow \def\apPPn##1{##1}%
1076:   \let\EXP=\apEPexp \def\LN{\apEPf{ln}}\let\SQRT=\apEPSqrt
1077:   \def\SIN{\apEPf{sin}}\def\COS{\apEPf{cos}}\def\TAN{\apEPf{tan}}%
1078:   \def\ASIN{\apEPf{arcsin}}\def\ACOS{\apEPf{arccos}}\def\ATAN{\apEPf{arctan}}%
1079:   \let\PI=\pi \def\PIhalf{{\pi\over2}}%
1080:   \let\ABS=\apEPabs \let\FAC=\apEPfac \let\BINOM=\apEPbinom

```

`\apATANox`: 46–47 `\ASIN`: 3, 4, 47 `\ACOS`: 3, 4, 47 `\eprint`: 7, 8, 47–50 `\apEpe`: 47, 49
`\apEPi`: 47–49 `\apEPj`: 48

```

1081: \let\SGN=\apEpsgn \let\iDIV=\apEPdiv \let\iMOD=\apEpmod
1082: \let\iFLOOR=\apEPifloor \let\iFRAC=\apEPifrac
1083: \let\apEPi=\relax \apEPj
1084: }
1085: \def\apEPj{}

```

All parameters are processed in new group. For example we have $\text{\apPLUS}\{a\}\{\text{\apDIV}\{b\}\{c\}\}$ in the \tmpb . Then the $\{a\}+\{\text{\apDIV}\{b\}\{c\}\}$ is processed and thus $\{a\}+\{\{b\}\over\{c\}\}$ is printed. The outer group is set by \eprint macro itself. So, the “printing” meaning of \apPLUS prepared in \apEPplus looks like:

```

1086: \def\apEPplus#1#2{\{#1\}+{\{#2\}\}}

```

apnum.tex

When we process the \tmpb with the output of the $\langle expression \rangle$ interpreter then the original positions of the round brackets are lost. We must to print these brackets if it is required by usual math syntax. For example $\text{\apMINUS}\{a\}\{\text{\apPLUS}\{b\}\{c\}\}$ must be printed as $a-(b+c)$. But $\text{\apMINUS}\{a\}\{\text{\apMUL}\{b\}\{c\}\}$ must be printed as $a-bc$.

The $\text{\apEP}\langle parameter \rangle \text{\empty}\text{\end}\langle a \rangle \langle b \rangle \langle c \rangle \langle d \rangle \langle e \rangle$ is used for this feature. The result of \apEP is the $\langle parameter \rangle$ enclosed or not enclosed in round brackets. It depends on the main operator M_{op} in the $\langle parameter \rangle$ and on the given parameters $\langle a \rangle \langle b \rangle \langle c \rangle \langle d \rangle \langle e \rangle$. If M_{op} is \apPLUS and $\langle a \rangle$ isn't dot or M_{op} is \apMINUS and $\langle b \rangle$ isn't dot or M_{op} is (unary minus or \apMUL) and $\langle c \rangle$ isn't dot or M_{op} is \apDIV and $\langle d \rangle$ isn't dot or M_{op} is \apPOW and $\langle e \rangle$ isn't dot then the $\langle parameter \rangle$ is enclosed in brackets using $\text{\left}\langle parameter \rangle\text{\right}$. Else the $\langle parameter \rangle$ is enclosed in invisible group-braces only. If M_{op} is nothing mentioned above (because single operand is here) then no brackets and no invisible braces are used and the $\langle parameter \rangle$ is printed “as is”. This feature is used in the printing version of \apMINUS , i. e. in \apEPminus macro (and in many others macros). The second parameter of \apMINUS is enclosed in brackets only if its main operator M_{op} is $+$ or $-$.

```

1087: \def\apEPminus#1#2{\{#1\}-\apEP#2\empty\end!!...}

```

apnum.tex

The unary minus in the cases like $-(a+b)$ are transformed to $\text{\apMUL}\{-1\}\{\text{\apPLUS}\{a\}\{b\}\}$ by the $\langle expression \rangle$ interpreter. But we don't need to print $-1\cdot(a+b)$. So, the printing version of \apMUL stored in the macro \apEpmul have an exception. First, we do the test, if #1 is equal to -1 . If this is true, then we print the unary minus else we print the whole first parameter enclosed in braces if its M_{op} is $+$ or $-$. The second parameter is enclosed in braces if its M_{op} is $+$ or $-$ or $*$. This needs more explanation: The multiplying chains as $a*b*c$ are processed from left to right in the $\langle expression \rangle$ scanner and the result is $\text{\apMUL}\{\text{\apMUL}\{a\}\{b\}\}\{c\}$. So, no brackets are printed. But the $a*-(b+c)$ is converted to $\text{\apMUL}\{a\}\{\text{\apMUL}\{-1\}\}\{\text{\apPLUS}\{b\}\{c\}\}$ and we need to print this as $a\cdot(-(b+c))$. This is the reason why the second parameter of \apMUL will be in brackets when its M_{op} is $*$.

```

1088: \def\apEpmul#1#2{\def\tmpa{#1}\def\tmpb{-1}%
1089: \ifx\tmpa\tmpb -\else \apEP#1\empty\end!!...\apMULop\fi
1090: \apEP#2\empty\end!!...}

```

apnum.tex

The \apEPdiv macro used for printing \apDIV is very easy. We needn't to set the outer group here because each parameter is enclosed in the group. We need not to add any round brackets here because fraction generated by \over is self explanatory from priority point of view. If you need to redefine \apEPdiv with the operator $/$ instead \over then you need to redefine \apEpmul too because you must enclose parameters with $M_{op} = \text{\apDIV}$ by brackets in such case.

```

1091: \def\apEPdiv#1#2{\{#1\}\over\{#2\}\}}

```

apnum.tex

The \apEppow macro used for printing \sim includes another speciality. When the base (the first $\langle parameter \rangle$) is a function-like macro \SIN , \COS etc. then we need to print $\text{\SIN}\{X\}^2$ as $\text{\sin}^2 x$. The test if the base is such special functions-like macro is performed by $\text{\apEppowa}\{\langle base \rangle\}\text{\end}\{\langle exponent \rangle\}$. If this is true then \apEppowa saves the $\langle exponent \rangle$ to the temporary macro \apEPy and only $\langle base \rangle$ is processed (the \apEPy is printed inside this processing) else \apEPy is empty and the $\langle base \rangle$ enclosed in brackets is followed by $\sim\{\langle exponent \rangle\}$. Note that the $\langle base \rangle$ isn't enclosed by brackets if its M_{op} is missing, i. e. the $\langle base \rangle$ is single operand.

\apEPplus : 47–49 \apEPminus : 47–49 \apEpmul : 47–49 \apEPdiv : 47–49 \apEppow : 47, 49
 \apEPy : 48–49

```

1092: \def\apEPpow#1#2{%
1093:   \let\apEPy=\empty \apEPpowa{#1}\end{#2}%
1094:   \ifx\apEPy\empty \apEPp#1\empty\end{!!!!~{#2}\else#1\fi
1095: }

```

The `\apEPpowa` macro detects the special function-like macro `\SIN`, `\COS` etc. by performing one expansion step on the tested $\langle base \rangle$. If the first $\langle token \rangle$ is `\apEPf` then the special function-like macro is detected. Note that `\SIN`, `\COS` etc. are defined as `\apEPf` in the `\apEpi` macro.

```

1096: \def\apEPpowa#1{\expandafter\apEPpowb#1;}
1097: \def\apEPpowb#1#2;\end{#3}\ifx#1\apEPf \def\apEPy{\let\apEPy=\empty #3}\fi}
1098: \let\apEPy=\empty

```

The functions like `\SIN{\langle expression \rangle}` are printed by `\apEPf {\langle name \rangle}{\langle expression \rangle}`. First, the `\mathop{\langle name \rangle}\nolimits` is printed. If `\apEPy` is non-empty then the exponent is printed by `~{\apEPy}`. Finally, the nested $\langle expression \rangle$ is printed by the nested `\eprint`.

```

1099: \def\apEPf#1#2{\mathop{\rm#1}\nolimits
1100:   \ifx\apEPy\empty \else ~{\apEPy}\let\apEPy=\empty \fi
1101:   \eprint{#2}{\expandafter\apEPb}%
1102: }

```

The code `\expandafter\apEPb` in the $\langle declaration \rangle$ part of `\eprint` expands the following `\tmpb` (the result of the $\langle expression \rangle$ scanner) and checks the first token. By default the $\langle expression \rangle$ will be enclosed by brackets (see the default `\next` definition where closing brace is printed by `\apEpe` macro used after expanded `\tmpb`). But if the first token is `\apPPn` or `\apDIV` then no brackets are printed around the $\langle expression \rangle$. Note that the $\langle expression \rangle$ scanner generates `\apPPn{\langle operand \rangle}` if and only if the whole $\langle expression \rangle$ is a single operand.

```

1103: \def\apEPb#1{\def\next{\left(\def\apEpe{\right)}}}%
1104:   \ifx\apPPn#1\let\next=\relax \fi
1105:   \ifx\apDIV#1\let\next=\relax \fi
1106:   \next#1%
1107: }

```

The meaning of `\apEPp` $\langle parameter \rangle \backslash \empty \backslash \text{end} \langle a \rangle \langle b \rangle \langle c \rangle \langle d \rangle \langle e \rangle$ is explained above, see the text where `\apEPminus` is introduced. Now, we focus to the implementation of this feature. The auxiliary macro `\apEPa` $\langle first token \rangle \{ \langle normal \rangle \} \{ \langle bracket \rangle \} \langle a \rangle \langle b \rangle \langle c \rangle \langle d \rangle \langle e \rangle$ is used twice: before processing the $\langle parameter \rangle$ #1#2 and after processing. The `\apEPa` inserts the $\langle normal \rangle$ or $\langle bracket \rangle$ depending on the condition described above where M_{op} is equal to the $\langle first token \rangle$. Note the trick with `\empty` which is inserted at the end of #2 parameter. The $\langle parameter \rangle$ should be in the form `\SIN{...}`. If the `\empty` token isn't added then #2 becomes the text without braces and this is not desired.

```

1108: \def\apEPp#1#2\end{#3#4#5#6#7}%
1109:   \apEPa#1\bggroup{\left(}{#3#4#5#6#7#1#2\apEPa#1\egroup{\right)}}{#3#4#5#6#7}
1110: \def\apEPa#1#2#3#4#5#6#7#8{%
1111:   \ifx#1\apEPplus \ifx#4.#2\else#3\fi\fi
1112:   \ifx#1\apEPminus \ifx#5.#2\else#3\fi\fi
1113:   \ifx#1\apEPmul \ifx#6.#2\else#3\fi\fi
1114:   \ifx#1\apEPdiv \ifx#7.#2\else#3\fi\fi
1115:   \ifx#1\apEPpow \ifx#8.#2\else#3\fi\fi
1116:   \ifx#1-\empty \ifx#6.#2\else#3\fi\fi
1117: }

```

The `\apMULop` is used as an operation mark for multiplying. It is `\cdot` by default but user can change this.

```

1118: \let\apMULop=\cdot

```

The single operand like 2.18 or `\X` or `\FAC{10}` is processed directly without any additional material. User can define “variables” as desired. The function-like macros provided by `apnum.tex` is initialized in `\apEpi` macro and the “printing macros” `\apEPabs`, `\apEPfac`, `\apEPbinom`, `\apEPsqrt`,

<code>\apEPpowa</code> : 48–49	<code>\apEPf</code> : 47, 49	<code>\apEPb</code> : 49–50	<code>\apEPp</code> : 48–49	<code>\apEPa</code> : 49
<code>\apMULop</code> : 8, 48–49	<code>\apEPabs</code> : 47, 50	<code>\apEPfac</code> : 47, 50	<code>\apEPbinom</code> : 47, 50	
<code>\apEPsqrt</code> : 47, 50				

`\apEexp`, `\apEsgn`, `\apEdivmod`, `\apEdiv`, `\apEimod`, `\apEifloor`, `\apEifrac` are defined here. The trick with `\expandafter\apEb` in the declaration part of the nested `\eprint` was explained above. Users can re-define these macros if they want.

apnum.tex

```

1119: \def\apEabs#1{\left|\eprint{#1}\right|}
1120: \def\apEfac#1{\eprint{#1}\expandafter\apEb\,}
1121: \def\apEbinom#1#2{{\eprint{#1}\choose\eprint{#2}}}
1122: \def\apESqrt#1{\sqrt{\eprint{#1}}}
1123: \def\apEexp#1{{\rm e}^{\eprint{#1}}}
1124: \def\apEsgn#1{\mathop{\rm sign}\eprint{#1}\expandafter\apEb}
1125: \def\apEdivmod#1#2#3{\left[\eprint{#2}\expandafter\apEb\%
1126: \mathbin{\rm #1}\eprint{#3}\expandafter\apEb\right]}
1127: \def\apEdiv{\apEdivmod{div}}
1128: \def\apEimod{\apEdivmod{mod}}
1129: \def\apEifloor#1{\left\lfloor\eprint{#1}\right\rfloor}
1130: \def\apEifrac#1{\left\{\eprint{#1}\right\}}

```

The `\corrnum` (*token*) macro expects *token* as a macro with number. It adds zero before decimal point if the sequence of *digits* before decimal point is empty. It uses a macro `\apEc` which works at expansion level. First, the occurrence of the `-` is tested. If it is true then `-` is expanded and the `\apEc` is called again. Else the zero is added if the first token is dot (this means if the *digits* before dot is empty).

apnum.tex

```

1132: \def\corrnum#1{\edef#1{\expandafter\apEc#1\end}}
1133: \def\apEc#1#2\end{\ifx#1-}\apEc#2\end\else\ifx#1.0.#2\else#1#2\fi\fi}

```

2.12 Conclusion

This code is here only for backward compatibility with old versions of `apnum.tex`. Don't use these sequences if you are implementing an internal feature because users can re-define these sequences.

apnum.tex

```

1137: \let\PLUS=\apPLUS \let\MINUS=\apMINUS \let\MUL=\apMUL \let\DIV=\apDIV \let\POW=\apPOW
1138: \let\SIGN=\apSIGN \let\ROUND=\apROUND \let\NORM=\apNORM \let\ROLL=\apROLL

```

Here is my little joke. Of course, this macro file works in LaTeX without problems because only TeX primitives (from classical TeX) and the `\newcount` macro are used here. But I wish to print my opinion about LaTeX. I hope that this doesn't matter and LaTeX users can use my macro because a typical LaTeX user doesn't read a terminal nor `.log` file.

apnum.tex

```

1140: \ifx\documentclass\undefined \else % please, don't remove this message
1141: \message{SORRY, you are using LaTeX. I don't recommend this. Petr Olsak}\fi
1142: \catcode'\@=\apnumZ
1143: \endinput

```

3 Index

The bold number is the number of the page where the item is documented. Other numbers are pagenumbers of the occurrences of such item. The items marked by \succ are mentioned in user documentation.

\succ <code>\ABS</code> : 37 , <u>3</u> , 5, 47	<code>\apDIGa</code> : 33 , 34
\succ <code>\ACOS</code> : 47 , <u>3</u> , 4	<code>\apDIGb</code> : 33 , 34
<code>\apADDzeros</code> : 35 , 16, 19, 23–24, 31	<code>\apDIGc</code> : 34 , 33
<code>\apATANox</code> : 47 , 46	<code>\apDIGd</code> : 34
<code>\apCOSx</code> : 46 , 44–45	<code>\apDIGe</code> : 34
<code>\apCOUNTS</code> : 36	<code>\apDIGf</code> : 34
<code>\apDIG</code> : 33 , 15–16, 18–19, 23–24, 29, 34, 39, 41, 46	\succ <code>\apDIV</code> : 23 , <u>6</u> , 11–12, 14, 24, 36–50
	<code>\apDIVa</code> : 23 , 24, 29

`\apEexp`: 47, 50 `\apEsgn`: 48, 50 `\apEdivmod`: 50 `\apEdiv`: 48, 50 `\apEimod`: 48, 50
`\apEifloor`: 48, 50 `\apEifrac`: 48, 50 `\corrnum`: 8, 50 `\apEc`: 50

- \apDIVcomp: 24, 23
- \apDIVcompA: 24, 25
- \apDIVcompB: 24, 25
- \apDIVg: 25, 23–24
- \apDIVh: 26, 25, 27
- \apDIVi: 26, 25
- \apDIVj: 26
- \apDIVp: 26
- \apDIVq: 26, 27
- \apDIVr: 27
- \apDIVt: 27
- \apDIVu: 27, 28
- \apDIVv: 28, 23–24
- \apDIVw: 28
- \apDIVxA: 26, 23–24, 27
- \apDIVxB: 26, 23–24
- ⋈\apE: 9, 4, 5–6, 10–14, 16, 18–19, 23–24, 28–29, 33, 37, 39, 41
- ⋈\apEadd: 33, 4, 5, 37
- \apEND: 33, 10, 12, 36–42, 44, 46–47
- ⋈\apEnum: 33, 4, 5, 37–38, 40, 45–47
- \apEPa: 49
- \apEPabs: 49, 47, 50
- \apEPb: 49, 50
- \apEPbinom: 49, 47, 50
- \apEPc: 50
- \apEPdiv: 48, 47, 49
- \apEPdivmod: 50
- \apEPe: 47, 49
- \apEPexp: 50, 47
- \apEPf: 49, 47
- \apEPfac: 49, 47, 50
- \apEPi: 47, 48–49
- \apEPidiv: 50, 48
- \apEPifloor: 50, 48
- \apEPifrac: 50, 48
- \apEPimod: 50, 48
- \apEPj: 47, 48
- \apEPminus: 48, 47, 49
- \apEPmul: 48, 47, 49
- \apEPp: 49, 48
- \apEPplus: 48, 47, 49
- \apEPpow: 48, 47, 49
- \apEPpowa: 49, 48
- \apEPsgn: 50, 48
- \apEPsqrt: 49, 47, 50
- \apEPy: 48, 49
- \apERR: 36, 24, 29, 37–39, 41, 46–47
- \apEVALa: 10, 12
- \apEVALb: 10, 11, 47
- \apEVALc: 10, 11
- \apEVALd: 10
- \apEVALdo: 12
- \apEVALe: 10
- \apEVALerror: 12, 11
- \apEVALf: 10
- \apEVALg: 10
- \apEVALh: 10
- \apEVALk: 11, 10
- \apEVALm: 11
- \apEVALn: 11, 10
- \apEVALo: 11, 10
- \apEVALp: 11, 10
- \apEVALpush: 12, 11
- \apEVALstack: 11, 12
- \apEVALxdo: 36
- ⋈\apFRAC: 9, 3, 4, 6, 24, 37, 39–47
- \apINIT: 36, 37–41, 44–47
- \apIVbase: 35, 17, 20–21, 27, 30
- \apIVdot: 35, 21, 27
- \apIVdotA: 35
- \apIVmod: 35, 15–16, 19, 24, 29
- \apIVread: 34, 16–17, 23–25
- \apIVreadA: 34
- \apIVreadX: 34, 23–24
- \apIVtrans: 34, 20–21, 30, 35
- \apIVwrite: 34, 18, 21, 23–24, 27–28, 30
- \apLNr: 42, 41
- \apLNra: 42
- \apLNrten: 42, 43
- \apLntaylor: 42, 41, 43
- \apLNten: 42, 43
- \apLNtenexec: 42, 41, 43
- ⋈\apMINUS: 15, 6, 11–12, 45, 47–48, 50
- ⋈\apMUL: 18, 6, 9–12, 14, 26, 36, 38, 40, 42, 44–45, 47–48, 50
- \apMULa: 18, 19, 28
- \apMULb: 20, 19, 28–29
- \apMULc: 20, 19
- \apMULd: 20, 19, 29
- \apMULE: 20, 21, 30
- \apMULf: 20, 21, 30
- \apMULg: 21, 19
- \apMULh: 21
- \apMULi: 21
- \apMULj: 21
- \apMULO: 21
- ⋈\apMULop: 49, 8, 48
- \apMULT: 21
- \apNL: 34, 16–17, 24–26
- \apNOPT: 37, 40, 42
- ⋈\apNORM: 30, 5, 14, 33, 50
- \apNORMa: 33, 30
- \apNORMb: 33
- \apNORMc: 33
- \apNORMd: 33
- \apNUMdigits: 35, 21, 27
- \apNUMdigitsA: 35
- \apOUTl: 36, 20, 26–27, 30
- \apOUTn: 36, 20–21, 27
- \apOUTs: 36, 20, 27
- \apOUTx: 36, 20–21, 27

- \apPI: 43, 44–45
- \apPIdigits: 43, 44
- \apPIexec: 43, 44–47
- \apPIexecA: 43
- \apPIexecB: 43, 44
- \apPIhalf: 43, 44–47
- \apPIvalue: 43, 44
- \apPLUS: 15, 6, 9–12, 14, 37–39, 41–48, 50
- \apPLUSa: 15, 16
- \apPLUSb: 16, 15
- \apPLUSc: 16, 17
- \apPLUSd: 17
- \apPLUSe: 16, 17
- \apPLUSf: 17
- \apPLUSg: 17, 15–16
- \apPLUSh: 17
- \apPLUSm: 17, 15–16
- \apPLUSp: 17, 15–16
- \apPLUSw: 18, 17
- \apPLUSxA: 15, 16–17
- \apPLUSxB: 15, 16–17
- \apPLUSxE: 18, 15–16
- \apPLUSy: 18, 16
- \apPLUSz: 18
- \apPOW: 28, 6, 12, 14, 36, 40, 42, 45, 47, 50
- \apPOWa: 28, 29–30
- \apPOWb: 29
- \apPOWd: 29
- \apPOWe: 30, 28–29
- \apPOWg: 30, 29
- \apPOWh: 30
- \apPOWn: 30, 29
- \apPOWna: 30
- \apPOWnn: 30
- \apPOWt: 30, 29
- \apPOWu: 30
- \apPOWv: 30
- \apPOWx: 28, 11–12, 47–48
- \apPPa: 12, 13–14
- \apPPab: 14, 15, 18, 24, 28–29, 33
- \apPPb: 12, 13–14
- \apPPc: 13
- \apPPd: 13
- \apPPe: 13
- \apPPf: 13
- \apPPg: 13
- \apPPh: 13
- \apPPi: 13
- \apPPj: 13
- \apPPk: 13, 14
- \apPPl: 13, 14
- \apPPm: 13, 14
- \apPPn: 14, 12, 47, 49
- \apPPs: 14, 18, 30–33
- \apPPt: 14
- \apPPu: 14
- \apREMDotR: 35, 27–28
- \apREMDotRa: 35
- \apREMfirst: 35, 6, 37, 40, 45–46
- \apREMzerosR: 35, 18, 27–28, 32
- \apREMzerosRa: 35
- \apREMzerosRb: 35
- \apRETURN: 36, 37–41, 46–47
- \apROLL: 30, 5, 14, 31, 33, 39, 41, 45, 50
- \apROLLa: 30, 18, 23–24, 29, 31–33
- \apROLLc: 31
- \apROLLd: 31
- \apROLLe: 31
- \apROLLf: 31
- \apROLLg: 31
- \apROLLh: 31
- \apROLLi: 31, 32
- \apROLLj: 32, 31
- \apROLLk: 32, 31
- \apROLLn: 32
- \apROLLo: 32
- \apROUND: 30, 5, 14, 32, 37–38, 40–43, 45–47, 50
- \apROUNDa: 32, 14, 30, 33
- \apROUNDb: 32
- \apROUNDc: 32
- \apROUNDd: 32
- \apROUNDe: 32, 33
- \apSIGN: 9, 4, 5–6, 10, 12–16, 18–19, 23–24, 29, 33, 37–42, 45–47, 50
- \apSINCOSa: 44, 45–46
- \apSINCOSo: 46, 44–45
- \apSINx: 46, 44–45
- \apSQRTr: 39
- \apSQRTra: 39, 40
- \apSQRTrb: 39, 40
- \apSQRTxo: 39, 44
- \apTAYLOR: 40, 41–42, 44–45, 47
- \apTESTdigit: 12, 10–11
- \apTOT: 9, 3, 4, 6, 24, 37, 43–46
- \apVERSION: 9
- \ASIN: 47, 3, 4
- \ATAN: 46, 3, 4, 47
- \BINOM: 38, 3, 47
- \corrnum: 50, 8
- \COS: 44, 3, 4, 45–49
- \do: 36, 38, 40–43, 45–47
- \eprint: 47, 7, 8, 48–50
- \evaldef: 10, 2, 3–6, 8–9, 12, 28, 36–41, 45–47
- \EXP: 40, 3, 4, 6, 41, 43, 47
- \FAC: 37, 3, 38, 47, 49
- \iDIV: 37, 3, 48
- \iFLOOR: 37, 3, 48
- \iFRAC: 37, 3, 48

>\iMOD: **37**, 3, 48
 >\LN: **41**, 3, 4, 6, 42, 47
 \localcounts: **36**, 37-41, 44-45, 47
 \loop: **37**, 38-40, 42, 44-45, 47
 >\OUT: **10**, 4, 5-6, 12-14, 16-21,
 23-24, 26-30, 33, 36-47
 >\PI: **44**, 3, 4, 47
 >\PIhalf: **44**, 3, 4, 47
 >\SGN: **37**, 3, 48
 >\SIN: **44**, 3, 4, 8, 45-49
 >\SQRT: **38**, 3, 4, 39, 44, 47
 >\TAN: **46**, 3, 4, 47
 >\XOUT: **28**, 6, 5, 14, 23-27, 32, 37-38, 45